5-2021

Coarse-Grained Model of Dielectric Geometry-Modified Screened Electrostatic Protein-Protein Interactions

Joshua H. Dickie
jd2012@rit.edu

COARSE-GRAINED MODEL OF DIELECTRIC GEOMETRY-MODIFIED SCREENED ELECTROSTATIC PROTEIN-PROTEIN INTERACTIONS

By

Joshua H. Dickie

A thesis submitted in partial fulfillment of the requirements for the degree of Masters in Physics, in the College of Science, Rochester Institute of Technology

May, 2021

Approved by _____

Prof. George M. Thurston Date

Director of Physics MS Program

SCHOOL OF PHYSICS AND ASTRONOMY

COLLEGE OF SCIENCE

ROCHESTER INSTITUTE OF TECHNOLOGY

ROCHESTER, NEW YORL

CERTIFICATE OF APPROVAL

MASTERS DEGREE THESIS

The Master of Science in Physics Thesis of Joshua H. Dickie has been examined and approved by the thesis committee as satisfactory for the thesis requirement for the Masters of Science in Physics Degree in the School of Physics and Astronomy at the Rochester Institute of Technology.

Dr. George M. Thurston, Thesis Advisor

Professor of Physics, RIT

Dr. Pratik P. Dholabhai

Assistant Professor of Physics, RIT

Committee Chair

Dr. David S. Ross

Professor of Mathematics, RIT

Dr. Joel D. Shore

Senior Lecturer in Physics, RIT

Date _____

Abstract

To refine a coarse-grained model of protein interactions, we seek to conveniently represent how dielectric interface geometry and charge placement affect screened aqueous electrostatic interactions. We study two neighboring low dielectric spheres with near-surface charges, for which we solve the linearized Poisson-Boltzmann equation as a function of sphere-sphere separation. The spheres have ~15Å diameters and internal static dielectric coefficients of 3. The solvent's Debye length is 6Å. These parameters are consistent with our charge-regulation model of bovine γB-Crystallin and with a wealth of previous experimental data for solutions of this protein. For a fixed on- or off-axis charge in the first sphere, the two-dimensional angular dependence of the near-surface potential in the second sphere is well-fit by a modified, rotated, possibly off-center Student t-distribution at each sphere-sphere distance. We use the full electrostatic solution to fit the parameters of these Student t-distributions as functions of sphere-sphere separation and angular placement of the charge in the first sphere. The approximation developed here is much more accurate than the unmodified Debye-Hückel screened potential and shows the potential for further refinement.

Contents

Abstract....2
List of Figures....4
Introduction....5
Methods: Model....6
Methods: Fitting Routine....8
The Approximation....9
Results: Other Dielectrics....10
Future Considerations....10
Charge Regulation....10
Additional Comparison to More Analytic Approaches....11
Extension to Dumbbell Geometry....11
Inverted Eiffel Towers....11
Conclusion & Implications....12
References....12
Appendix I: PDE Solver....14
Appendix II: Mathematica Programs....19
Appendix III: The Quadratic Interpolator....51

List of Figures

Figure		Page
1	**Left:** *Visualization of bovine GammaB-Crystallin protein[6].* **Right:** *Simplified dumbbell geometry of bovine GammaB-Crystallin (PDB ID 1amm). The spheres have a radius of ~14.6 Å while the red (negative), blue (positive), and black (neutral) dots indicate the charge at a given protonation site for the modelled most-probable protonation state[2].*	6
2	*Illustration of the model simplification of the electrostatics used here. The spheres on the right are used to replace the lower sphere of the left dumbbell and the upper sphere of the right dumbbell.*	6
3	*This is a diagram of how the position of each charge is parameterized in the model (see text). The angle that the charge makes from the line joining the low dielectric sphere centers, and the distance between the surfaces of the spheres are the two input variables for our PDE computations. The outermost grey sphere is the salt exclusion zone, which has a depth of 1.4 Å. The inner boundary of the lighter colored shells indicates the radius where model protein charges will be placed. Surrounding and i**n**-between the spheres is an aqueous solution modeled as an electrolyte with a static dielectric coefficient of 78.5 containing ions and having a Debye length of 6.0 Å.*	7
4	*Example depicting PDE solution. The innermost sphere on each side shows the radii at which all charges are placed. The sphere immediately around each of these spheres is the low dielectric 'surface' of the protein at radius ~14.6 Å. The labelled red surfaces are electrostatic equipotentials.*	7
5	*The two spheres at left are identical to those in figure 4. The orange points are a constant distance from the center of the right-hand sphere. They show the points at which we calculated the potential using a quadratic interpolator when these points were not on the grid of our PDE solver. The projection of the orange points into a 2D plane allows us to represent the potential as height (at right). Note that a line (dotted black) drawn from the charge to the center of the second sphere passes almost directly through the peak of the potential (see red line) on the buried charge sphere.*	8
6	*Electrostatic potential energy surface from figure 5 right, rotated for clarity. The potential energy is comparable to the thermal energy over much of the modeled surface for charge placement. The blue surface shows the corresponding Student t-Distribution based fit; the extent of agreement shown is typical (see text).*	8
7	*Dimensionless peak amplitude (see figure 6) versus sphere surface separation and angle of input charge off axis. The orange points show amplitudes from the best fit student t-distribution for each angle and distance. The blue surface shows the fit given by equation Eq. (2).*	9
8	*Calculated dimensionless peak potential minus the approximated potential according to Eq.(2). Note that the difference reaches its maximum of* $\approx 0.6eV/kT$ *at* $\approx$ 2.5 *Å.*	9
9	*This is a cross section through the center of the spheres which shows contours like those in Figure 3, but instead using an inner dielectric coefficient of 4. Again, the low dielectric distorts the electrostatic potential contours so as to extend further from the charge in the first sphere.*	10
10	Buried on-axis dimensionless potential in second sphere vs sphere-sphere separation distance for an on-axis charge in the first sphere. The orange points are the modelled data points, and the solid blue line is our approximation. The dashed line shows the dimensionless potential that would instead result from using the Debye-Hückel equation for an aqueous electrolyte.	11

Introduction

Protein-protein interactions are a vital area of research for many diseases. Mutations of proteins can cause a variety of diseases and we do not yet have a predictive physics-based algorithm for determining if a mutation will be important. As protein-protein interactions cannot easily be represented with sufficient accuracy analytically, numerical methods will be essential to get results accurate enough to be used for prediction. However, numerical solutions of many facets of protein-protein interactions would require prohibitively long computation times (even with the use of supercomputers) to get the needed results.

Our primary interest here is in the protein human $\gamma D-$ Crystallin found in the eye's lens. However, the homologous bovine $\gamma B-$ Crystallin has far more experimental data and is far easier to study in the lab. Here we investigate the electrostatic contributions to the 'potential energy landscape'. To do so we needed to develop an approximation method to the numerical solution of the model PDE we are using because of limited computation time available to us.

$\gamma B - Crystallin$ undergoes a liquid-liquid phase transition which has been implicated in cataract formation. Single point mutations have been previously examined and it has been shown that these single mutations can cause changes in the phase diagram[1]. Our approximation method focuses on the electrostatic interactions $\gamma B - Crystallin$ has with other identical $\gamma B - Crystallin$ proteins in an electrolyte solution. By using a numerical PDE solver (discussed below) we were able to solve a coarse-grained model of the $\gamma B - Crystallin$ interaction energy map for a large variety of relative orientations and distances. By then interpolating between these solutions, we built an approximation which can be used to explore the full electrostatic potential energy space of interaction.

The electrostatic interactions of proteins in electrolyte solution are only influential at extremely close distances ($\approx\leq 9$Å for electrolytes with a Debye length of 6Å). When compared to the size of $\gamma B - Crystallin$ (≈ 45Å long by 28Å wide and deep) this distance is significantly smaller than the protein itself. Restricting our analysis to the regions which have a noticeable effect on the potential energy landscape allows us to run more permutations and build out a higher resolution for the potential energy landscape. In addition, by using the linearized Poisson-Boltzmann equation we are able to analyze the system one charge contribution at a time. This dramatically improves the speed of computation because of the many charges on each protein.

$\gamma B - Crystallin$ has tens of thousands of protonation patterns which it can inhabit. However, at ambient temperatures it has been modelled that only 500 of them constitute over 97% of the probability space[2]. This can happen because of the exchange of protons with the environment. The primary factors which govern this exchange are the proton affinities of each titratable amino acid side chain, and the pH of the solution. The more complicated secondary factors have to do with the electrostatic interactions between all the charged sites on both a given protein and on neighboring proteins.

By setting up a PDE solution that accounted for both the low dielectric protein interiors and the screening, we found that the Debye-Hückel screening form for a uniform aqueous electrolyte[3] made for substantial underestimates. Specifically,

the standard form can underestimate the calculated electrostatic potential at the location of neighboring protein charges by up to a factor of 4. Our current fitting form to the PDE solution has greatly reduced this error as discussed below.

Single mutations of $\gamma - Crystallin$ can cause cataracts[1] and so we need predictive physics models to know which mutations are harmful. By understanding the energy landscape of $\gamma B - Crystallin$ we aspire to predict quantitatively how single point mutations affect strong attractive wells that appear in this landscape. These strong attractive wells have been implicated in the clumping of $\gamma - Crystallin$[4]; aggregation of eye lens proteins is one known underlying cause of cataracts[5]. Therefore, we propose that by making an accurate mapping of the potential energy landscape of this protein we will improve our ability to predict how each amino acid contributes to the total potential, which will let us predict which mutations are harmful.

This paper will start by discussing the geometry simplifications we have made for our model and then move on to the PDE solver which we used to calculate the electrostatic potential. Then we will discuss the approximation method used to reach our analytic approximation, after which we will present our current results and best fit. We will then move to edge cases and possible avenues of further research before finishing up with final thoughts and acknowledgements. For a detailed explanation and instruction on the specific codes used please refer to the appendix where the code will be explained in exhaustive detail.

Methods: Model

There are many forces which operate on proteins in solution. It is too complicated to model all these possible interactions at once and so we have performed the following simplifications. We will only be considering the electrostatic interactions from the charges and the dielectric. In addition, we will approximate the shape of the protein as two spheres stuck together (see Figure 1).

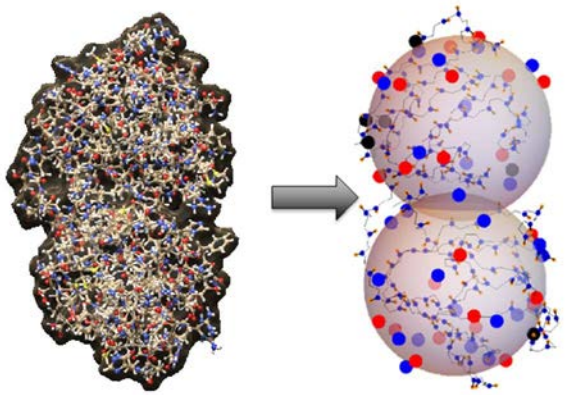


Figure 1
Left: *Visualization of bovine GammaB-Crystallin protein*[6].
Right: *Simplified dumbbell geometry of bovine GammaB-Crystallin (PDB ID 1amm). The spheres have a radius of ~14.6 Å while the red (negative), blue (positive), and black (neutral) dots indicate the charge at a given protonation site for the modelled most-probable protonation state*[2].

From the single dumbbell geometry, we then consider two dumbbells in close proximity (see Figure 2). By considering only one sphere from each dumbbell at a time we can add the contributions from each charge by solving a linearized Poisson-Boltzmann equation.

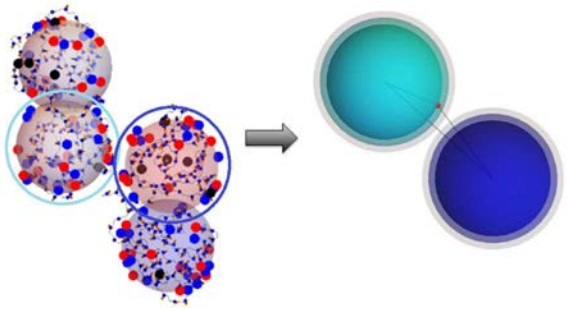


Figure 2: *Illustration of the model simplification of the electrostatics used here. The spheres on the right are used to replace the lower sphere of the left dumbbell and the upper sphere of the right dumbbell.*

Let us consider one of the spheres of our simplified-geometry protein. Placing a charge 1.5 Å inside the sphere simulates the charge depth in our model proteins. We then include a salt-exclusion zone around the protein (see Figure 3). If we place an identical protein in close proximity, we can also consider one of its spheres. Now we can examine how charges on one protein affect the potential inside the other, which will eventually let us compose the energy landscape of this protein.

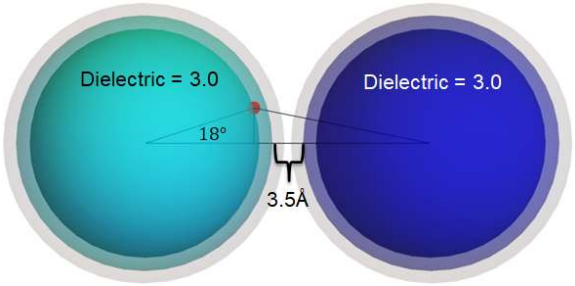


Figure 3: *This is a diagram of how the position of each charge is parameterized in the model (see text). The angle that the charge makes from the line joining the low dielectric sphere centers, and the distance between the surfaces of the spheres are the two input variables for our PDE computations. The outermost grey sphere is the salt exclusion zone, which has a depth of 1.4 Å. The inner boundary of the lighter colored shells indicates the radius where model protein charges will be placed. Surrounding and in-between the spheres is an aqueous solution modeled as an electrolyte with a static dielectric coefficient of 78.5 containing ions and having a Debye length of 6.0 Å.*

We note that the potential energy landscape for the interaction of any two non-spherically symmetric molecules is six-dimensional. This set of dimensions can be described as follows. Here, one dimension is the distance between the dumbbells. The second is the 'twist' the dumbbells have with respect to each other. The third and fourth are the polar and azimuthal angles which describe the position of the second dumbbell relative to the first. The fifth and sixth dimensions are the polar and azimuthal angles which describe which part of the second dumbbell is pointed toward the center of the first.

Because we are using a linear PDE model for the potential, we can construct the electrostatic interaction from two charges at a time and then sum them. In Figure 4, you can see the surfaces of equipotential leaving the first protein and entering the second one.

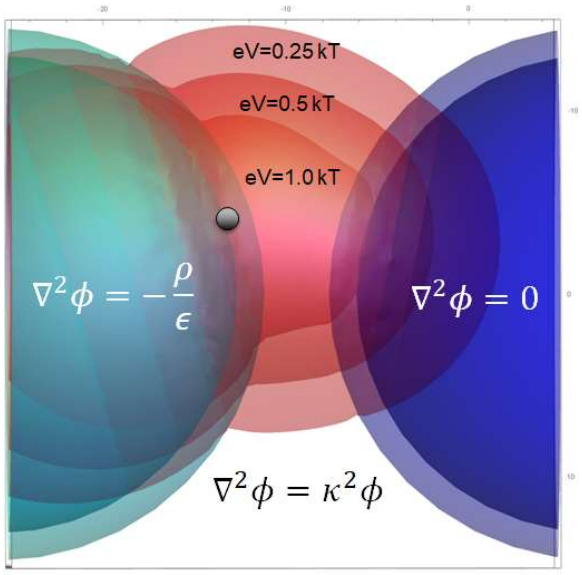


Figure 4: *Example depicting PDE solution. The innermost sphere on each side shows the radii at which all charges are placed. The sphere immediately around each of these spheres is the low dielectric 'surface' of the protein at radius ~14.6 Å. The labelled red surfaces are electrostatic equipotentials.*

The choices of Debye length and dielectric coefficient were based on the experimental buffers that have been used and a fit to the potentiometric titration curve[2]. Specifically, it was found that a dielectric coefficient of 3.0 yielded the best fit to the experimentally inferred charge vs pH curve. The Debye length of 6Å corresponds to that of the buffer used in the majority of the phase diagram and light-scattering experiments done on bovine $\gamma\mathrm{B}-\mathrm{Crystallin}$[2,7–15]. These choices will facilitate direct comparisons to experiment.

To create an approximation for the electrostatic potential in and between the two dielectric spheres we solve the equations

$$\nabla^2\phi = -\frac{\rho}{\epsilon}, \nabla^2\phi = \kappa^2\phi, \nabla^2\phi = 0$$

within their respective regions of the first sphere, the electrolyte solution, and the second sphere.

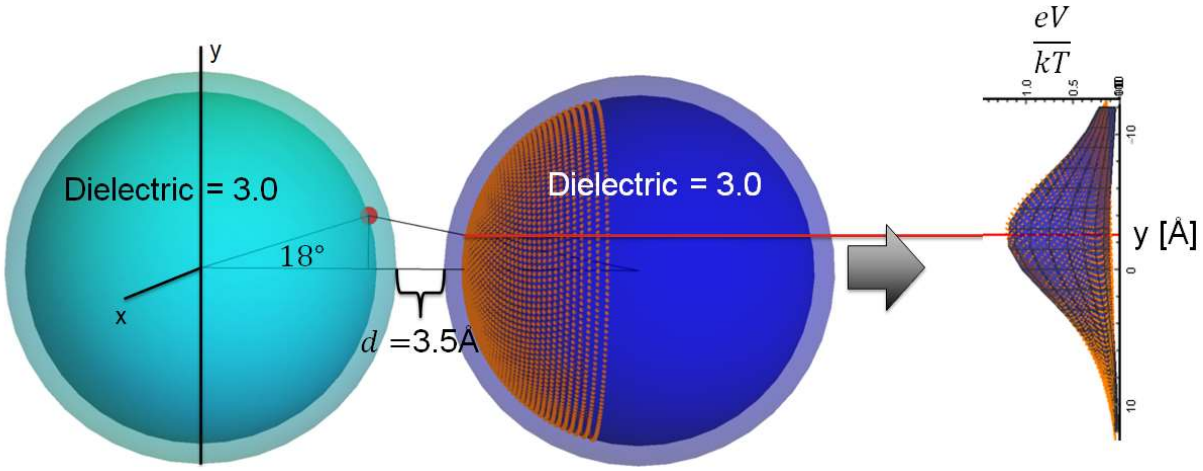


Figure 5: *The two spheres at left are identical to those in figure 4. The orange points are a constant distance from the center of the right-hand sphere. They show the points at which we calculated the potential using a quadratic interpolator when these points were not on the grid of our PDE solver. The projection of the orange points into a 2D plane allows us to represent the potential as height (at right). Note that a line (dotted black) drawn from the charge to the center of the second sphere passes almost directly through the peak of the potential (see red line) on the buried charge sphere.*

The potential energy function we get by solving the above equations must be of high enough resolution that our primary goal of creating an informative 6-dimensional energy landscape can be realized. For this reason, we have chosen to use a numerical PDE solver with a grid length of 0.5Å. The criterion for the over-relaxation method to terminate is that two successive iterations have a maximum change in their potential of 0.01%.

We also used a reflected boundary condition for the PDE solver. To combat the numerical drift that can occur from using this boundary condition, we attempted to keep the number of cycles low and were successful in this as none of the orientations required more than 500 cycles to stabilize.

We used the successive over-relaxation method for the PDE solver with an over-relaxation parameter of 1.8. Picking a practical value of the over-relaxation parameter is delicate as the rate of convergence changes rapidly near the maximum convergence rate[16]. A relaxation factor of 2 or more will yield an unstable non-converging algorithm.

Methods: Fitting Routine

After running the numerical PDE solver, we sampled the potential along a sphere inside the second protein (illustrated by the orange points in figure 5) to obtain a plot of the potential energy for one electronic charge, divided by the thermal energy, as is shown on the right. To interpolate between the grid points of our PDE solver we used a quadratic interpolator (discussed below and in the appendix). Note how the peak of our data lies close to a line drawn from the charge on the first sphere to the center of the second.

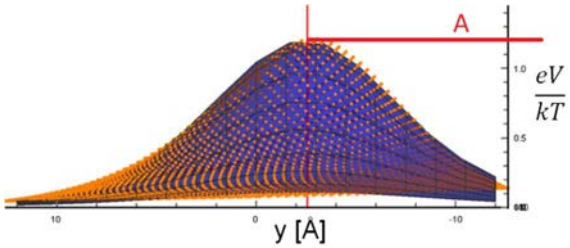


Figure 6: *Electrostatic potential energy surface from figure 5 right, rotated for clarity. The potential energy is comparable to the thermal energy over much of the modeled surface for charge placement. The blue surface shows the corresponding Student t-Distribution based fit; the extent of agreement shown is typical (see text).*

The buried points of the second sphere can be projected down onto a two-dimensional plane to which we assign the third dimension to the potential. The electrostatic potential evaluated at these points is shown in both Figure 5 and above such a plane in Figure 6.

We selected a peak cutoff potential of 0.5eV/kT. Any distance and/or angle combination which failed to reach this energy was excluded from the fit of our approximation. This choice was motivated by the fact that the electrostatic potential energy contributes to a Boltzmann factor. This means that the contributions to the partition function of orientations and distances with a higher ratio of electrostatic energy to thermal energy will be exponentially larger than those which have a small ($\sim \leq 1$) ratio.

The quadratic interpolator used is a contribution of Dr. John Hamilton[17] and is described in Appendix III.

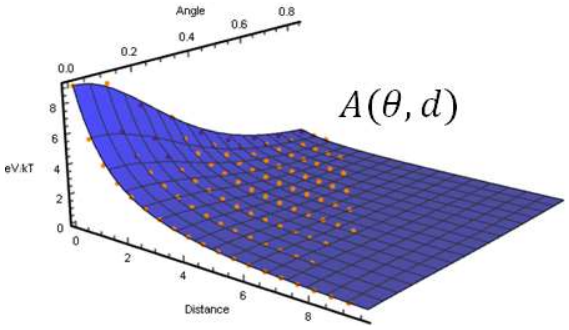


Figure 76: *Dimensionless peak amplitude (see figure 6) versus sphere surface separation and angle of input charge off axis. The orange points show amplitudes from the best fit student t-distribution for each angle and distance. The blue surface shows the fit given by equation Eq. (2).*

The Approximation

Here we present our current working model for the potential along the interior of the second sphere as a function of the distance between the spheres (d) and the angle (θ) which the charge makes with respect to a line joining the centers of the two spheres (see figure 5).

$$\frac{Potential}{kT} = A\left(\frac{\nu}{\nu + \frac{x^2 + (y-c)^2}{\sigma^2}}\right)^{\frac{1+\nu}{2}} \quad (1)$$

$$A(\theta, d) = \frac{543.1\left(\frac{1}{2.423 + 9.704\theta^2}\right)^{1.711}}{(3. + d)^{2.337}} \quad (2)$$

$$\sigma(\theta, d) = 3.256 + 0.04816\theta + 3.402\theta^2 + 0.7172d - 0.395\,\theta\,d - 0.0542d^2 \quad (3)$$

$$\nu(\theta, d) = 2.965 - 1.525\,\theta + 5.76\theta^2 + 0.2802\,d - 1.196\,\theta\,d + 0.2418\,d^2 \quad (4)$$

The value of c is derived by drawing a line between the charge on the first sphere to the middle of the second sphere. Then project the point at which the line intersects the interior-colored sphere (where the charges are placed) onto the xy-plane. The value the point has along the y axis is the value of c for that distance and charge angle.

Our current model fits from $0 \leq \theta \leq \frac{3\pi}{10}$ and $0\text{Å} \leq d \leq 9\text{Å}$ (see Figure 7) with a maximum error (shown in Figure 8) of approximately 0.6 eV/kT, which occurs at $d = 2.5\text{Å}$ and $\theta = \pi/30$. However, we note that this value of d is smaller than the approximate diameter of one water molecule (2.8Å) so that if at least one layer of water molecules were to be between the spheres, as expected, the maximum relevant error would be smaller.

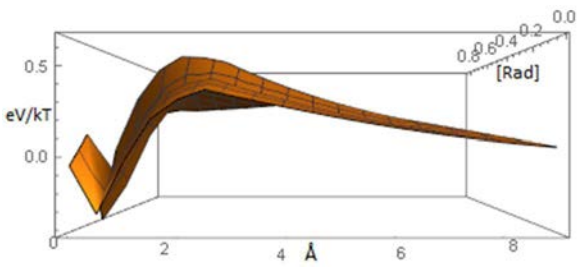


Figure 8: *Calculated dimensionless peak potential minus the approximated potential according to Eq.(2). Note that the difference reaches its maximum of* $\approx 0.6eV/kT$ *at* $\approx 2.5\,Å$.

By using the above functions, we are able to approximate the energy landscape of $\gamma B - Crystallin$ when in proximity to another $\gamma B - Crystallin$.

A note of caution should be made about the appearance of the Student t-distribution in our approximation. The use of the Student t-distribution is unrelated to its statistical context. We have instead used it because it has fewer parameters than two gaussians and fits the data more accurately at the fringes. The appearance of the Student t-distribution in the amplitude function is likely due to a symmetry in our formulation.

Our current approximation is accurate for when the second sphere is either very close or very far away from the first. However, there is a notable 'bump' in the residual map at distance between 1Å and 6Å (see figure 8). This residual peaks at 2.5Å and we are exploring the possibility that this is related to the fact that the salt-exclusion zone extends 1.4Å from the surface of each sphere in our model.

Results: Other Dielectrics

After acquiring the approximation for the two spheres with dielectric coefficients of 3, we wanted to see how the approximation varied as the dielectric was changed. The impetus for this is the opportunity to extend the approximation to other proteins which may be modelled by different interior dielectrics. In doing this we found that the Student t-distribution still provided a good fit and that the parameters needed were very similar to those for the dielectric 3 case. Figure 9 shows an example using an inner dielectric coefficient of 4. We are currently working on extending the approximation quantitively to other dielectric values.

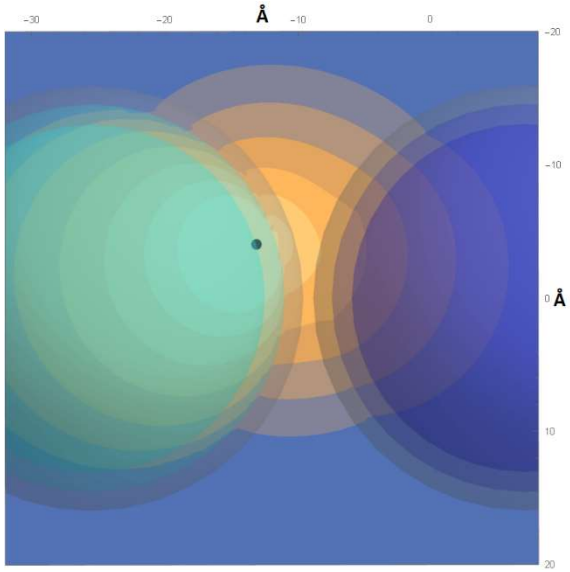


Figure 9: *This is a cross section through the center of the spheres which shows contours like those in Figure 3, but instead using an inner dielectric coefficient of 4. Again, the low dielectric distorts the electrostatic potential contours so as to extend further from the charge in the first sphere.*

Future Considerations

There are many avenues which can be further explored based on the work we have done. Charge regulation, extending the approximation to the dumbbell geometry, and locating and categorizing the deep attractive wells present in the six-dimensional potential energy space are all extremely important. Below we will briefly outline the benefits this further research would give, as well as the difficulties we foresee.

Charge Regulation

$\gamma B - Crystallin$ protonation patterns depend on the local electrostatic potential[2]. Use of our approximation will enable us to evaluate the effect of different pairs of protonation patterns on the interaction between two neighboring proteins.

Additional Comparison to More Analytic Approaches

The work of Levin, Li, and Fisher considered low dielectric spheres containing centrally located charges immersed in an electrolyte solution and provided analytic expressions for the electrostatic potential and the interaction energy of the dielectric spheres[18]. Whereas our charges are not centrally located, their work will provide for comparison with suitably constructed solutions of our model PDE.

Very recently published work of Yu has considerably extended the work of Levin, Li, and Fisher to apply to more arbitrary charge distributions in the low dielectric spheres, as well as extension to N neighboring spheres[19]. We expect that this work will also provide for further tests and potential insight into our approximation.

Extension to Dumbbell Geometry

The dumbbell geometry was chosen because of its similarity to the shape of the full $\gamma B - Crystallin$ protein (see Figure 1). As noted above, by using the dumbbell, we are working to construct an approximation of the 6-dimensional potential energy space of two neighboring $\gamma B - Crystallin$ molecules. This is crucial for understanding the molecular basis of the observed liquid-liquid phase separation and light scattering[2,4,7–11,20]. The goal is to have a detailed model for how individual features of the protein affect the phase boundary.

The size and positioning of the assumed spherical dumbbell's 'lobes' were determined by minimizing the RMS error on the position of the charges the protein has when projected onto the spherical surfaces. The major benefit to this method is that the smoothness of the dumbbell allows us to run through a large number of relative orientations with the protein-spheres using the approximation we created above.

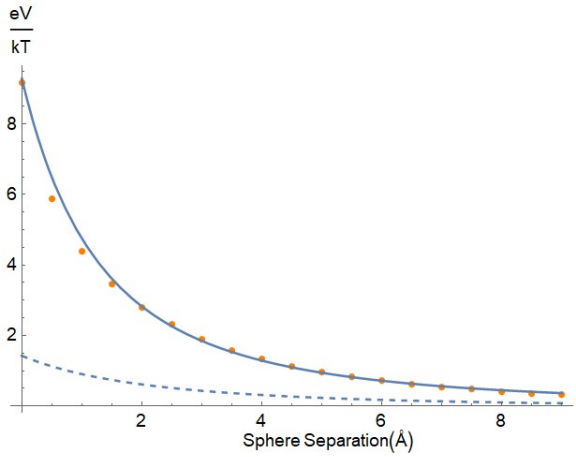


Figure 10: Buried on-axis dimensionless potential in second sphere vs sphere-sphere separation distance for an on-axis charge in the first sphere. The orange points are the modelled data points, and the solid blue line is our approximation. The dashed line shows the dimensionless potential that would instead result from using the Debye-Hückel equation for an aqueous electrolyte.

By rerunning the PDE solver for the two dumbbells we can compare how well the approximation fits the dumbbell data. It is our expectation that the introduction of the second sphere for each protein could require a correction term to the approximation. However, it is possible that such a correction term would be smaller than the errors we are already accepting as part of our simplified premise.

Inverted Eiffel Towers

Within the 6-dimensional energy landscape of $\gamma B - Crystallin$, previous work has indicated evidence of deep attractive potentials which can overwhelm the thermal energy of the protein and cause two proteins to 'lock' into their orientation[4]. This phenomenon is important in governing the shape of the phase diagram, and in particular the liquid-liquid separation boundary.

A relevant consideration is that the electrostatic approximation described in this paper is accurate to within 0.6 eV/kT, which is substantially

improved from the unmodified Debye-Hückel screened potential, which can deviate from values in our model by up to $8\ eV/kT$ (see Figure 10). With this substantial increase in accuracy, we expect the current approximation to be more useful in identifying the potential energy locks in the potential energy space of $\gamma B - Crystallin$. In addition, the benefit of using a linear equation here is that we can not only more easily model the energy landscape of a normal $\gamma B - Crystallin$ but also the energy landscape of a mutated $\gamma B - Crystallin$. This could allow us to quantitively predict which mutations of charged amino acids will affect protein interactions and therefore the liquid-liquid separation boundary in the phase diagram.

Conclusion & Implications

The approximation detailed above has proven to be accurate within the constraints of our original simplifications and will be exceedingly quick to calculate when used for large-scale investigations of protein interactions. While an approximation can never be exact, and neither is the full PDE for that matter[3], time requirements impose a severe limitation to the number of relative orientations and neighboring proteins which can be analyzed with direct use of the PDE. We expect that further work using the present approximation will be invaluable for understanding protein mutations and the role they can play in disease.

References

(1) Pande, A.; Pande, J.; Asherie, N.; Lomakin, A.; Ogun, O.; King, J. A.; Lubsen, N. H.; Walton, D.; Benedek, G. B. Molecular Basis of a Progressive Juvenile-Onset Hereditary Cataract. *Proc. Natl. Acad. Sci.* **2000**, *97* (5), 1993–1998. https://doi.org/10.1073/pnas.040554397.

(2) Wahle, C. W.; Martini, K. M.; Hollenbeck, D. M.; Langner, A.; Ross, D. S.; Hamilton, J. F.; Thurston, G. M. Model for Screened, Charge-Regulated Electrostatics of an Eye Lens Protein: Bovine GammaB-Crystallin. *Phys. Rev. E* **2017**, *96* (3), 032415. https://doi.org/10.1103/PhysRevE.96.032415.

(3) McQuarrie, D. *Statistical Mechanics*.

(4) Lomakin, A.; Asherie, N.; Benedek, G. B. Aeolotopic Interactions of Globular Proteins. *Proc. Natl. Acad. Sci.* **1999**, *96* (17), 9465–9468. https://doi.org/10.1073/pnas.96.17.9465.

(5) Benedek, G. B.; Pande, J.; Thurston, G. M.; Clark, J. I. Theoretical and Experimental Basis for the Inhibition of Cataract. *Prog. Retin. Eye Res.* **1999**, *18* (3), 391–402. https://doi.org/10.1016/S1350-9462(98)00023-8.

(6) *Molecular Graphics and Analyses Performed with UCSF Chimera, Developed by the Resource for Biocomputing, Visualization, and Informatics at the University of California, San Francisco, with Support from NIH P41-GM103311.*

(7) Thomson, J. A.; Schurtenberger, P.; Thurston, G. M.; Benedek, G. B. Binary Liquid Phase Separation and Critical Phenomena in a Protein/Water Solution. *Proc. Natl. Acad. Sci.* **1987**, *84* (20), 7079–7083. https://doi.org/10.1073/pnas.84.20.7079.

(8) Schurtenberger, P.; Chamberlin, R. A.; Thurston, G. M.; Thomson, J. A.; Benedek, G. B. Observation of Critical Phenomena in a Protein-Water Solution. *Phys Rev Lett* **1989**, *63* (19), 2064–2067. https://doi.org/10.1103/PhysRevLett.63.2064.

(9) Berland, C. R.; Thurston, G. M.; Kondo, M.; Broide, M. L.; Pande, J.; Ogun, O.; Benedek, G. B. Solid-Liquid Phase Boundaries of Lens Protein Solutions. *Proc. Natl. Acad. Sci.* **1992**, *89* (4), 1214–1218. https://doi.org/10.1073/pnas.89.4.1214.

(10) Broide, M. L.; Berland, C. R.; Pande, J.; Ogun, O. O.; Benedek, G. B. Binary-Liquid Phase Separation of Lens Protein Solutions. *Proc.*

Natl. Acad. Sci. **1991**, *88* (13), 5660–5664. https://doi.org/10.1073/pnas.88.13.5660.
(11) Fine, B. M.; Lomakin, A.; Ogun, O. O.; Benedek, G. B. Static Structure Factor and Collective Diffusion of Globular Proteins in Concentrated Aqueous Solution. *J. Chem. Phys.* **1996**, *104* (1), 326–335. https://doi.org/10.1063/1.470904.
(12) Stradner, A.; Thurston, G. M.; Lobaskin, V.; Schurtenberger, P. Structure and Interactions of Lens Proteins in Dilute and Concentrated Solutions. *Prog Colloid Polym Sci*. https://doi.org/10.1088/0953-8984/17/31/005.
(13) Stradner; Foffi, G.; Dorsaz, N.; Thurston, G. M.; Schurtenberger, P. New Insight into Cataract Formation: Enhanced Stability through Mutual Attraction. *Phys. Rev. Lett. 99 19 Art No 198103*.
(14) Dorsaz, N.; Thurston, G. M.; Stradner, A.; Schurtenberger, P.; Foffi, G. Colloidal Characterization and Thermodynamic Stability of Binary Eye Lens Protein Mixtures. *J Phys Chem*. https://doi.org/10.1021/jp807103f.
(15) Dorsaz, N.; Thurston, G. M.; Stradner, A.; Schurtenberger, P. Phase Separation in Binary Eye Lens Protein Mixtures. *Soft Matter 71763-1776 2011*. https://doi.org/10.1039/C0SM00156B.
(16) Stoer, J.; Bulirsch, R. *Introduction to Numerical Analysis*; Springer, 2002.
(17) Personal Correspondence with J. Hamilton.
(18) Levin, Y.; Li, X.; Fisher, M. E. Coulombic Criticality in General Dimensions. *Phys. Rev. Lett.* **1994**, *73* (20), 2716–2719. https://doi.org/10.1103/PhysRevLett.73.2716.
(19) Yu, Y.-K. Electrostatics of Charged Dielectric Spheres with Application to Biological Systems. III. Rigorous Ionic Screening at the Debye-Hückel Level. *Phys. Rev. E* **2020**, *102* (5), 052404. https://doi.org/10.1103/PhysRevE.102.052404.
(20) Quinn, M. K.; James, S.; McManus, J. J. Chemical Modification Alters Protein–Protein Interactions and Can Lead to Lower Protein Solubility. *J. Phys. Chem. B* **2019**, acs.jpcb.9b02368. https://doi.org/10.1021/acs.jpcb.9b02368.

Appendix I: PDE Solver

Our PDE solver was created in the Fortran programming language and recompiled using the GNU Fortran compiler, it is attached as a separate file to this paper. There are six subroutines which are called in the main program of twosphereelectro. They are, in order:

- parameters
- initial
- setq
- solve
- output_WOC
- output_potential

These subroutines have self-explanatory names, however, because of the number of variables used, a detailed description of each variable must be given.

The following variables are defined in the main program:

t0: 8 byte real, contains initial start time of the program.

idx0: integer, contains starting separation of charges placed inside the proteins at the buried depth. Multiplied by dx(1) for actual distance. dx(1) has a default value of 0.5 angstroms.

idxf: integer, contains ending separation of charges. Multiplied by dx(1) for actual final distance (dx(1) has a default value of 0.5 angstroms).

isph: integer, determines which sphere has the charge on it. 1 is for the immobile sphere and 2 is for the one which is moved each iteration.

lc: integer, contains charge index for the array which will contain where all the charges are placed. lc=1 here indicates that only one real charge is present.

dist: 8 byte real. The distance between the two charges in the PDE solver (one real, one fictitious).

The following variables are defined in the subroutine "parameters":

ein: 8 byte real, contains user input interior dielectric coefficient for the two proteins (both protein spheres must have the same dielectric coefficient in this PDE solver).

chargeAngle: 8 byte real, contains user input off axis angle of the charged point on the first sphere. The angle is taken from a line which connects to the two centers of the spheres in the PDE solver. Note this value is multiplied by $\pi/30$ and so an input value of '3' would result in an angle off axis of $3\pi/30$.

nx, ny, nz: integers, contains number of grid points along each axis (x,y,z) for the region to be solved.

xlen: 8 byte real of 3 dimensions. Contains the length in angstroms of each side of the box (x,y,z).

dx: 8 byte real of 3 dimensions, contains length along each axis of the subdivisions of the box. Calculated by taking the length of the box in each dimension and dividing by the number of grid points minus 1 in that respective dimension.

dyzx, dzxy, dxyz: 8 byte reals. Contains surface area of the subdivisions along the size dictated by the first two letters ad divided by the length of the third e.g. dyzx = (dy * dz) / dx. Used in calculating the dielectric coefficient along the surface boundary of each subdivision. The division by the third dimension is necessary as earlier in the code which deals with setting up the dielectric coefficient of each cell, we get the dielectric coefficient of the box. By multiplying that result by the constants here we are then

able to get the surface dielectric of each box from the total internal dielectric value.

radius: 8 byte real in 2x2 dimensions. The radius of the protein is placed in the first vector (1:2,1) and the thickness of the salt exclusion zone is placed in the second (1:2,2). Note this allows the spheres to be of unequal size and have different salt exclusion zones but in all uses of this code we used we had them equal.

eout: 8 byte real. Contains the dielectric coefficient of the dielectric outside the protein-spheres. Default is 78.5.

debye: 8 byte real. Contains the Debye length of the external aqueous solution in angstroms. Default is 6 Angstroms.

temp: 8 byte real. Contains the temperature of the solution in the solver, default is 300 degrees kelvin.

om: 8 byte real. Contains the over-relaxation parameter which is $\in [1,2]$. Default is 1.85.

tol: 8 byte real. Contains the maximum difference between two successive iterations of the relaxation procedure before a solution can be accepted. Default is $5 * 10^{-4}$.

itmax: integer. Contains the maximum number of iterations the PDE solver can attempt for a particular distance. Default is 10,000.

omm1: 8 byte real. Contains the value 1-om. Used to multiply the potential in the last iteration for the current iteration. Default is -0.85.

electroncharge: 8 byte real. Contains the charge of an electron is standard units ($1.60217733000 * 10^{-19}$.

eps0: 8 byte real. Contains the permittivity of free space, default value is $8.85418781762 * 10^{-12}$.

kB: 8 byte real. Contains the Boltzmann constant of $1.38065800000 * 10^{-23}$.

angstrom: 8 byte real. Conversion of meters to angstroms, default value is 10^{-10}.

bjerrum: 8 byte real. Contains the scale factor we multiply by at the end of the program to get the potential in the unitless quantity of eV/kT. Calculated with: $\frac{electroncharg^{\ 2}}{angstrom*eps0*kB*temp}$.

The following variables are defined in the subroutine "inital":

Pi: 8 byte real. Contains the value of π as far as is possible for 8 bytes.

chargeLabels: 4 characters variable. Contains "Cl". Not relevant for actual calculations.

charge: 8 byte real with dimension 2 by the number of charges. Default is 2x1. Contains a value of 1 for (1,1). All charges are expressed in integer values (single electron charges from charged amino acids).

chargePos: 8 byte real with dimension 2x(# of charges)x3. The "2" is the number of spheres. The number of charges is the second (default is 1). The final dimension is the (x,y,z) dimensions for each charge position.

charge: 8 byte real with dimensions of 2x(# of charges). Contains the magnitude and sign of the charges. the first dimension divides the charge based on which sphere they are in; the second dimension is for each charge.

center: 8 byte real with dimensions of 2x3. The first dimension is indexed for the two spheres and the second is for each axis the center is on.

icp: integer with dimensions of 2x(# of charges)x3. This array contains the position of each charge on each protein-sphere. The first dimension is the sphere index, the second is the

charge index, and the final dimension is for the axis index.

ex,ey,ez: 8 byte real with dimensions of nx,ny,nz which by default is (241,141,141) but one dimension (the dimension of the second character in the variable name) is increased by 1 to include the outer surface of the cells on the edge of the box. These arrays are the same size as the box which the simulation is taking place in but is offset by a distance half the length of a cell. The second character in the variable name indicates which dimension the array is offset in. These arrays contain the dielectric coefficient on the surface of each cell.

eps: 8 byte real. This is the sum of the dielectric surfaces of a cell divided by six.

kap: 8 byte real of dimensions of nx,ny,nz which by default is (241,141,141). This array is created by multiplying the volume of a cell by "eps" and then dividing by 'debye'.

The following variables are defined in the subroutine "setq":

q: 8 byte real with dimensions of 2x(# of charges)x3. The first dimension is for the index of each protein sphere, the second is for the index of each charge, and the final dimension holds the index of the cell position.

The following variables are defined in the subroutine "solve":

potential: 8 byte real with dimensions of nx,ny,nz which by default is (241,141,141). Contains the total potential of the box, operated on in the solve subroutine.

test: 8 byte real. Contains the largest percent change a single cells potential has had during the update step of the iteration.

pin,pjn,pkn,pip,pjp,pkp: 8 byte real. The first character of the variable represents "potential". the second character is either i,j, or k and represents the index along the respective dimension. The final character is either "n" or "p" and means either negative or positive, indicating which direction the potential is being taken from. All together this means that 'pin' indicates the potential in the cell in the negative x direction. This is used in the loop for the iteration.

sig: 8 byte real. Contains the dielectric surface weighted average of the potential of all nearby cells through the faces of one cell. Used for each cell during each iteration.

p0min: 8 byte real. Contains a number used to prevent infinities from appearing while calculating the change in potential from one iteration to the next. Default value is 10^{-10}.

relerr: 8 byte real. Contains the relative error of the current cells potential when compared to its previous potential.

The following variables are defined in the subroutine "output_WOC":

work: 8 byte real. Contains the potential at the fictious charge point on the second sphere.

Now that we have identified all the variables used we can discuss how to use the twosphereelectro program and how it functions.

When the program is first called the command line will prompt the user to enter the input the dielectric coefficient to be used for the interior of the proteins. Then the user will be prompted to enter what angle (in units of $\pi/30$) the charge on the first sphere makes with the second. As the potential is symmetric about the axis the

angle of the charge is always taken to be in the y direction.

Next cpu_time is called t0 is defined. After which the subroutine 'parameters' is called. This subroutine contains all of the fundamental parameters which the program will be calling. Inside of this subroutine very few calculations are made, but any changes made here will affect how the physical laws are expressed and solved for in the program. One thing to note is that while the dielectric coefficient inside of the spheres is taken from user input, the exterior dielectric as well as the Debye length of the aqueous solution is set here.

Next, we allocate the computer memory necessary for the variables 'potential', 'kap', 'q', 'ex', 'ey', and 'ez'. Note that 'nx','ny', and 'nz' were define in parameters and have a default value of (241,141,141). Once these arrays have been allocated in memory the main programs loop begins. We call 'initial' which generates the dielectric arrays and the charge array. The dielectric arrays are the most complex of the arrays generated in this program as each one is offset by half a cell length in each respective dimension. This is so that the dielectric constant at the surface of each cell can be used to weight the potential which is drawn from each neighboring cell in the 'solve' subroutine which is to come.

The dielectric surfaces are themselves calculated by simply checking if the center of each surface falls within the bounds of one of the spheres. If it is it has a value of 'ein' if not, 'eout'. We then find the scaled salt contributions at every surface by once again checking the position of each surface against the two spheres with one correction. If we are within the 'radius(1:2,1)' of either sphere there is no salt, but further if we are within 'radius(1:2,1)' + 'radius(1:2,2)' we also have no salt because we are then within the 'no salt region' of our model. Then if we are in a region with salt we calculate the average dielectric coefficient of each cell's surfaces and set 'kap' of the same index as the cell to this value multiplied by the volume of the cell and divided by the Debye length. Finally, 'initial' comes to an end by scaling 'ex','ey','ez' by their respective surface areas divided by the third dimension.

The next subroutine to be called is 'solve'. As the name suggests this is the subroutine in which the PDE is solved. We immediately initialize the potential array to 0 and then enter a do loop which proceeds along the x direction, then the y, then the z. We take the potential of all six adjacent cells and multiply each one by the dielectric surface coefficient we calculated previously. If we are on the edge of the box we 'reflect' the potential, taking the potential through the surface which borders the edge of the box to be equal to the potential through the opposite face. Doing this we assign this value to 'sig' and then proceed to the potential update step. Via the over-relation method we multiply the potential that used to be in this cell by 'omm1'. Then we add to this 'om' times the new weighted average potential, plus any charges present in the cell we are evaluating, then divide this by the 'eps' of the cell (which is the average of the dielectric values along the faces) plus the salt contribution (if any).

After calculating the new potential we check what the percent difference between the old potential and the new potential is. If the previous error is smaller than 'p0min' we instead use 'p0min' for calculating the percent difference in order to prevent division by 0 errors. If the percent difference is found to be largest that the previous largest percent difference of this iteration it is replaced.

After an iteration is performed two checks take place. The first is if the largest percent change in the potential is smaller than the 'tol' (tolerance' which we defined in 'parameters'. If it is we then check if there have been more than 20 iterations.

If so we have found the solution to our PDE. The loop is then exited and the potential array is scaled by 'bjerrum' which puts our values into the unitless quantity eV/kT.

After the potential has been found the data is exported via 'output_WOC' and 'output_potential'. These are both simple programs. The important one of the two, and the one used in our paper, is the 'output_potential' subroutine. This saves out the potential array cell by cell with the cells laid out with each value printed on a new line and in the order of the x dimension, then the y, and finally the z. The file is saved out to the same location as the main program, and so when generating data using this program it is advisable to move the main program to the location you want the data.

Appendix II - Mathematica Programs

Below are all the Mathematica programs used to create the approximation of the PDE solutions detailed in the paper.

Rebuild Files As Interior of Second Protein

This program takes as input the files generated by twosphereelectro and outputs the interpolated potential along the buried depth surface on the second sphere out to an angle of 60 degrees from a line joining the centers of the two spheres. The generated files are saved out under a subfolder called "Condensed" and separated by angle.

Load These Functions First

In[ ]:=
```
gapSizesForGeneral = (Range[51] / 2.) // Drop[#, 6] & // (# - 3.5) &;

suffixesForGeneral = gapSizesForGeneral + startpoint //
       Map[(ToString[#] // StringReplace[#, "." → "p"] &) &, #] & //
     Map[Characters[#] &, #] & //
    Map[(If[#[[-1]] == "p", Append[#, "0"], #]) &, #] & // Map[StringJoin, #] &;
```

In[ ]:=
```
xDom = Range[1, 141];
yDom = Range[1, 141];
zDom = Range[1, 241];

uMax = Length[xDom];
vMax = Length[yDom];
wMax = Length[zDom];

iDom = Range[uMax];
jDom = Range[vMax];
kDom = Range[wMax];
```

In[ ]:=
```
Clear[uTest, vTest, wTest];

uTest[u_] := (2 ≤ u) && (u < uMax);
vTest[v_] := (2 ≤ v) && (v < vMax);
wTest[w_] := (2 ≤ w) && (w < wMax);
```

In[ ]:=
```
importPartitionButDontPutInCoordinates[filePath_] :=
 Import[filePath, "Table"] // Flatten // Partition[#, 241] & //
  Partition[#, 141] & (* 241 makes soda straws and 141 makes pancakes *)
```

In[ ]:=
```
filepaths[folderName_] :=
  {Map[StringJoin[mainFolderLocation, folderName, "/dist=", #, ".txt"] &,
     suffixesForGeneral[[Range[txtToTextPoint]]]],
    Map[StringJoin[mainFolderLocation, folderName, "/dist=", #, ".text"] &,
     suffixesForGeneral // Drop[#, txtToTextPoint] &]} // Flatten;
```

In[•]:=
```
withMaxAnglesuccessiveCirclesCenteredOnCenterToCenterAxisSamplingFunction[
  partitionedFile_, gapSizeInAngstroms_, {thetaSampleSpacing_,
   maxAngleFromPole_, approximateNumberOfEquatorialPhiSamples_},
  angleOffAxisOfCharge_, sphereRadius_, distanceInsideOfEmbeddedCharge_] :=
 Module[{justPartitioned, successiveCirclestableOfValues,
   xyPlaneprojectedTableOfValues, valuesInOrder, overallSum, maxValue,
   cumulativeDistribution, conjoinedAnglesWithCDF, handyFunction,
   centerAngleEstimate, upperPartWithinOneStdDevOfASingleGaussian,
   oneStdDevSingleGaussian, studentTDistributionFit,
   toBePlottedStudentT, toBePlottedStudentTResiduals},
  (*justPartitioned = importPartitionButDontPutInCoordinates[
     filePathThatHasDottxtAppended];*)

  successiveCirclestableOfValues =
   Table[{theta, phi, quadraticInterpolator[71 + (2 * 13.0751) *
         Sin[theta] * Cos[phi], 71 + (2 * 13.0751) * Sin[theta] * Sin[phi],
       96(* grid point that a centered charge would have had *) + 3(* grid
         point value for edge of charged sphere *) + (gapSizeInAngstroms * 2)
        (*to account for the number of GRID SPACINGS taken up by the
         minimum gap between the spheres for a given file *) + 2 * 14.5751 -
        (2 * 13.0751) * Cos[theta], partitionedFile(*justPartitioned*)]},
     {theta, 0.001, maxAngleFromPole + 0.001, thetaSampleSpacing}, {phi, 0,
      2 * Pi, (2 * Pi / (Sin[theta] * approximateNumberOfEquatorialPhiSamples))}] //
    Flatten[#, 1] &;
  (* will give just one value at the theta = 0, provided
    Sin[initial value of theta]*approximateNumberOfEquatorialPhiSamples < 1 *)
  (*But the phiSampleSpacing should be made, I think,
  to depend on the the value of theta  -
   take a number of samples that is proportional to Sin[theta].  I
    think I need to do this with a Table of Tables instead.  No,
  that was not necessary.  Above works OK for now. *)

  xyPlaneprojectedTableOfValues =
   Map[({sphereRadius * Sin[#[[1]]] * Cos[#[[2]]], sphereRadius * Sin[#[[1]]] *
        Sin[#[[2]]], #[[3]]}) &, successiveCirclestableOfValues];

  Return[xyPlaneprojectedTableOfValues]
 ]
```

In[ ]:=
```
successiveCirclesCenteredOnCenterToCenterAxisSamplingFunction[
  partitionedFile_, gapSizeInAngstroms_,
  {thetaSampleSpacing_, approximateNumberOfEquatorialPhiSamples_},
  angleOffAxisOfCharge_, sphereRadius_, distanceInsideOfEmbeddedCharge_] :=
 Module[{justPartitioned, successiveCirclestableOfValues,
   xyPlaneprojectedTableOfValues, valuesInOrder, overallSum, maxValue,
   cumulativeDistribution, conjoinedAnglesWithCDF, handyFunction,
   centerAngleEstimate, upperPartWithinOneStdDevOfASingleGaussian,
   oneStdDevSingleGaussian, studentTDistributionFit,
   toBePlottedStudentT, toBePlottedStudentTResiduals},
  (*justPartitioned = importPartitionButDontPutInCoordinates[
     filePathThatHasDottxtAppended];*)

  successiveCirclestableOfValues =
   Table[{theta, phi, quadraticInterpolator[71 + (2 * 13.0751) *
         Sin[theta] * Cos[phi], 71 + (2 * 13.0751) * Sin[theta] * Sin[phi],
       96(* grid point that a centered charge would have had *) + 3(* grid
         point value for edge of charged sphere *) + (gapSizeInAngstroms * 2)
        (*to account for the number of GRID SPACINGS taken up by the
         minimum gap between the spheres for a given file *) + 2 * 14.5751 -
        (2 * 13.0751) * Cos[theta], partitionedFile(*justPartitioned*)]},
     {theta, 0.001, Pi / 2 + 0.001, thetaSampleSpacing}, {phi, 0, 2 * Pi,
      (2 * Pi / (Sin[theta] * approximateNumberOfEquatorialPhiSamples))}] //
    Flatten[#, 1] &;
  (* will give just one value at the theta = 0, provided
    Sin[initial value of theta]*approximateNumberOfEquatorialPhiSamples < 1 *)
  (*But the phiSampleSpacing should be made, I think,
  to depend on the the value of theta  -
   take a number of samples that is proportional to Sin[theta].  I
    think I need to do this with a Table of Tables instead.  No,
  that was not necessary.  Above works OK for now. *)

  xyPlaneprojectedTableOfValues =
   Map[({sphereRadius * Sin[#[[1]]] * Cos[#[[2]]], sphereRadius * Sin[#[[1]]] *
        Sin[#[[2]]], #[[3]]}) &, successiveCirclestableOfValues];

  Return[{successiveCirclestableOfValues, xyPlaneprojectedTableOfValues}]
 ]
```

In[ ]:=
```
quadraticInterpolator[u_, v_, w_, tbl_] :=
  Module[{okay, i0, j0, k0, i1, j1, k1, r, s, t, im, jm, km, ip, jp, kp,
    wgti, wgtj, wgtk, value, tmp1, tmp2, fxx, vxx, fyy, vyy, fzz, vzz},
```

```
okay = uTest[u] && vTest[v] && wTest[w];
If[! okay, Return["Domain Fault"]];

i0 = u // Floor;
j0 = v // Floor;
k0 = w // Floor;
i1 = i0 + 1;
j1 = j0 + 1;
k1 = k0 + 1;
r = 2 (u - i0) - 1;
s = 2 (v - j0) - 1;
t = 2 (w - k0) - 1;
im = i0 - 1;
jm = j0 - 1;
km = k0 - 1;

wgti = {1 - r, 1 + r} / 2.;
wgtj = {1 - s, 1 + s} / 2.;
wgtk = {1 - t, 1 + t} / 2.;

(* TriLinear Value *)
tmp1 = Table[
  tbl[[im + i, jm + j, km + k]] × wgti[[i]] × wgtj[[j]] × wgtk[[k]]
  , {i, 1, 2}, {j, 1, 2}, {k, 1, 2}];
value = tmp1 // Flatten // Total;

(* Second Order Differences and Quadratic Adjustments *)
ip = i1 + 1; (* direction i *)
tmp2 = tbl[[im ;; ip, j0 ;; j1, k0 ;; k1]];
fxx = (tmp2[[1, ;; , ;;]] - tmp2[[2, ;; , ;;]] -
    tmp2[[3, ;; , ;;]] + tmp2[[4, ;; , ;;]]) / 8.;
fxx = fxx (1 - r^2) / 2.;
tmp2 = Table[
  fxx[[j, k]] × wgtj[[j]] × wgtk[[k]]
  , {j, 1, 2}, {k, 1, 2}];
vxx = tmp2 // Flatten // Total; (* Quadratic Adjustment for X *)

jp = j1 + 1; (* direction j *)
tmp2 = tbl[[i0 ;; i1, jm ;; jp, k0 ;; k1]];
fyy = (tmp2[[ ;; , 1, ;;]] - tmp2[[ ;; , 2, ;;]] -
    tmp2[[ ;; , 3, ;;]] + tmp2[[ ;; , 4, ;;]]) / 8.;
fyy = fyy (1 - s^2) / 2.;
tmp2 = Table[
```

```
    fyy[[i, k]] × wgti[[i]] × wgtk[[k]]
    , {i, 1, 2}, {k, 1, 2}];
  vyy = tmp2 // Flatten // Total; (* Quadratic Adjustment for Y *)

  kp = k1 + 1; (* direction k *)
  tmp2 = tbl[[i0 ;; i1, j0 ;; j1, km ;; kp]];
  fzz = (tmp2[[ ;; , ;; , 1]] - tmp2[[ ;; , ;; , 2]] -
      tmp2[[ ;; , ;; , 3]] + tmp2[[ ;; , ;; , 4]]) / 8.;
  fzz = fzz (1 - t^2) / 2.;
  tmp2 = Table[
    fzz[[i, j]] × wgti[[i]] × wgtj[[j]]
    , {i, 1, 2}, {j, 1, 2}];
  vzz = tmp2 // Flatten // Total; (* Quadratic Adjustment for Z *)

  value - vxx - vyy - vzz
 ];
```

Variables to Edit

After loading in the functions above we can proceed to analyzing the output of twosphereelectro. First, confirm that the number of folders which are going to be read in is correct. The program always starts at the 0 angle folder and proceeds up to "numberOfFoldersToReadIn"*π/30 in increments of π/30.

```
numberOfFoldersToReadIn = 11;
folderLocation = "E:\\Research Data\\Dielectric Value of 3\\";
```

Rebuild files

```
progress = 0;
ProgressIndicator[
 Dynamic[progress / ((Length[FileNames[]]) * numberOfFilesToReadIn)]]
SetDirectory[];
Do[
 Print[
  "Working on folder: " <> "OffCenter" <> ToString[folderNum - 1] <> "PiOver30"];
 SetDirectory[folderLocation <> "OffCenter" <>
   ToString[folderNum - 1] <> "PiOver30"];
 CreateDirectory[folderLocation <> "/Condensed/OffCenter" <>
   ToString[i] <> "PiOver30"];
 allFiles = FileNames["dist=*"];
 allFiles = {allFiles[[
     Length[allFiles] - Total[StringCount[allFiles, "txt"]] + 1 ;;]], allFiles[[
      ;; Length[allFiles] - Total[StringCount[allFiles, "txt"]]]]} // Flatten;
 CreateDirectory[folderLocation <> "OffCenter" <>
   ToString[folderNum - 1] <> "PiOver30" <> "/Condensed"];
 Do[
  progress = progress + 1;
  fileToCheck = {importPartitionButDontPutInCoordinates[
     folderLocation <> "OffCenter" <> ToString[folderNum - 1] <>
      "PiOver30" <> "/" <> allFiles[[fileNum]]], (folderNum - 1) / 2};
  fileToCheck = {fileToCheck};
  interiorSpherePotential =
   Map[withMaxAnglesuccessiveCirclesCenteredOnCenterToCenterAxisSamplingFunction[
     #[[1]], (fileNum - 1) * 0.5 (*gapSizeInAngstroms*),
     {Pi / 120 (*thetaSampleSpacing_*), Pi / 3 (*maxAngleFromPole_*), 200
       (*approximateNumberOfEquatorialPhiSamples_*)}, (folderNum * Pi / 30)
     (*angleOffAxisOfCharge_*), 14.5751 (*sphereRadius_*), 1.5
     (*distanceInsideOfEmbeddedCharge_*)] &, fileToCheck];
  interiorSpherePotential = interiorSpherePotential // Flatten //
    Partition[#, 3] &;
  Export[folderLocation <> "/Condensed/OffCenter" <>
    ToString[folderNum - 1] <> "PiOver30" <> "/Condensed/Condensed" <>
    StringSplit[allFiles[[fileNum]], "."][[1]] <> ".tsv", interiorSpherePotential];

  , {fileNum, 1, Length[FileNames[]]}]

 , {folderNum, 1, numberOfFoldersToReadIn }]
```

Compile Condensed Fitting Data

After generating the condensed data files you can then compile the individual files into a form which is used for the other programs. Each distance is appended to a master file which contains all the interpolated points at each distance. Check that you have input the correct distance to read out to and that the root director for the angle folders is correct. The output of this program is a series of files which are saved to a file located in 'folderPath' and named after the angle of the folder they are compiled from and the distance the compilation goes out to.

Variables to check/change

```
readOutToDistance = 10. ;(*In Angstroms*)
readStartAngle = 0; (*Will be appended with π/30 during runtime*)
readOutToAngle = 10; (*Will be appended with π/30 during runtime*)
folderPath = "E:\\Research Data\\Dielectric Value of 4\\Condensed Points";
(*Location of the files to be compiled*)
```

Compile Interpolated Files

```
dataStuff = List[readOutToAngle - readStartAngle, readOutToDistance * 2];
SetDirectory[folderPath];
compiledData = List[];
progress = 0;
ProgressIndicator[
 Dynamic[progress / (readOutToDistance * (readOutToAngle - readStartAngle))],
 Indeterminate]

Do[
 SetDirectory[folderPath <> "/OffCenter" <> ToString[i] <> "PiOver30"];
 fileList = FileNames["Condensed*.tsv"];
 (*Print[fileList];*)
 listTotal = Count[fileList, "*=??p?.tsv"];
 totalCount = 0;

 Do[totalCount =
    totalCount + StringCount[fileList[[k]], "=" ~~ _ ~~ "p" ~~ _ ~~ ".tsv"];
  , {k, 1, Length[fileList]}];

 fileList = {fileList[[Length[fileList] - totalCount + 1 ;;]],
    fileList[[ ;; Length[fileList] - totalCount]]} // Flatten;
 Do[
  progress = progress + 1;
  newData = List[];
  fileData = Import[folderPath <> "/OffCenter" <>
     ToString[i ] <> "PiOver30/" <> ToString[fileList[[j]]]];
  Do[
   AppendTo[newData, {i * Pi / 30, (j - 1) / 2 , fileData[[q]]} // Flatten]
   , {q, 1, Length[fileData]}];

  AppendTo[compiledData , newData];

  , {j, 1, readOutToDistance * 2 + 1}];

 compiledData = compiledData // Flatten // Partition[#, 5] &;
 Export["Condensed" <> ToString[i] <> "PiOver30All.tsv", compiledData];
 compiledData = List[];
 , {i, readStartAngle, readOutToAngle}]
```

Generate cutOffPoints.wl

This program will generate the file 'cutOffPoints.wl' which is needed to remove peaks relative orientations that are too far to significantly affect the potential energy landscape.

Functions To Load

```
importPartitionButDontPutInCoordinates[filePath_] :=
  Import[filePath, "Table"] // Flatten // Partition[#, 241] & // Partition[#, 141] &
 (* 241 makes soda straws and 141 makes pancakes *);
```

In[ ]:=
```
withMaxAnglesuccessiveCirclesCenteredOnCenterToCenterAxisSamplingFunction[
  partitionedFile_, gapSizeInAngstroms_, {thetaSampleSpacing_,
   maxAngleFromPole_, approximateNumberOfEquatorialPhiSamples_},
  angleOffAxisOfCharge_, sphereRadius_, distanceInsideOfEmbeddedCharge_] :=
 Module[{justPartitioned, successiveCirclestableOfValues,
   xyPlaneprojectedTableOfValues, valuesInOrder, overallSum, maxValue,
   cumulativeDistribution, conjoinedAnglesWithCDF, handyFunction,
   centerAngleEstimate, upperPartWithinOneStdDevOfASingleGaussian,
   oneStdDevSingleGaussian, studentTDistributionFit,
   toBePlottedStudentT, toBePlottedStudentTResiduals},
  (*justPartitioned = importPartitionButDontPutInCoordinates[
     filePathThatHasDottxtAppended];*)

  successiveCirclestableOfValues =
   Table[{theta, phi, quadraticInterpolator[71 + (2 * 13.0751) *
         Sin[theta] * Cos[phi], 71 + (2 * 13.0751) * Sin[theta] * Sin[phi],
       96(* grid point that a centered charge would have had *) + 3(* grid
         point value for edge of charged sphere *) + (gapSizeInAngstroms * 2)
        (*to account for the number of GRID SPACINGS taken up by the
         minimum gap between the spheres for a given file *) + 2 * 14.5751 -
        (2 * 13.0751) * Cos[theta], partitionedFile(*justPartitioned*)]},
     {theta, 0.001, maxAngleFromPole + 0.001, thetaSampleSpacing}, {phi, 0,
      2 * Pi, (2 * Pi / (Sin[theta] * approximateNumberOfEquatorialPhiSamples))}] //
    Flatten[#, 1] &;
  (* will give just one value at the theta = 0, provided
    Sin[initial value of theta]*approximateNumberOfEquatorialPhiSamples < 1 *)
  (*But the phiSampleSpacing should be made, I think,
  to depend on the the value of theta  -
   take a number of samples that is proportional to Sin[theta].  I
    think I need to do this with a Table of Tables instead.  No,
  that was not necessary.  Above works OK for now. *)

  xyPlaneprojectedTableOfValues =
   Map[({sphereRadius * Sin[#[[1]]] * Cos[#[[2]]], sphereRadius * Sin[#[[1]]] *
        Sin[#[[2]]], #[[3]]}) &, successiveCirclestableOfValues];

  Return[xyPlaneprojectedTableOfValues]
 ]
```

In[ ]:=
```
quadraticInterpolator[u_, v_, w_, tbl_] :=
  Module[{okay, i0, j0, k0, i1, j1, k1, r, s, t, im, jm, km, ip, jp, kp,
    wgti, wgtj, wgtk, value, tmp1, tmp2, fxx, vxx, fyy, vyy, fzz, vzz},
```

```
okay = uTest[u] && vTest[v] && wTest[w];
If[! okay, Return["Domain Fault"]];

i0 = u // Floor;
j0 = v // Floor;
k0 = w // Floor;
i1 = i0 + 1;
j1 = j0 + 1;
k1 = k0 + 1;
r = 2 (u - i0) - 1;
s = 2 (v - j0) - 1;
t = 2 (w - k0) - 1;
im = i0 - 1;
jm = j0 - 1;
km = k0 - 1;

wgti = {1 - r, 1 + r} / 2.;
wgtj = {1 - s, 1 + s} / 2.;
wgtk = {1 - t, 1 + t} / 2.;

(* TriLinear Value *)
tmp1 = Table[
  tbl[[im + i, jm + j, km + k]] × wgti[[i]] × wgtj[[j]] × wgtk[[k]]
  , {i, 1, 2}, {j, 1, 2}, {k, 1, 2}];
value = tmp1 // Flatten // Total;

(* Second Order Differences and Quadratic Adjustments *)
ip = i1 + 1; (* direction i *)
tmp2 = tbl[[im ;; ip, j0 ;; j1, k0 ;; k1]];
fxx = (tmp2[[1, ;; , ;;]] - tmp2[[2, ;; , ;;]] -
    tmp2[[3, ;; , ;;]] + tmp2[[4, ;; , ;;]]) / 8.;
fxx = fxx (1 - r^2) / 2.;
tmp2 = Table[
  fxx[[j, k]] × wgtj[[j]] × wgtk[[k]]
  , {j, 1, 2}, {k, 1, 2}];
vxx = tmp2 // Flatten // Total; (* Quadratic Adjustment for X *)

jp = j1 + 1; (* direction j *)
tmp2 = tbl[[i0 ;; i1, jm ;; jp, k0 ;; k1]];
fyy = (tmp2[[ ;; , 1, ;;]] - tmp2[[ ;; , 2, ;;]] -
    tmp2[[ ;; , 3, ;;]] + tmp2[[ ;; , 4, ;;]]) / 8.;
fyy = fyy (1 - s^2) / 2.;
tmp2 = Table[
```

```
    fyy[[i, k]] × wgti[[i]] × wgtk[[k]]
    , {i, 1, 2}, {k, 1, 2}];
  vyy = tmp2 // Flatten // Total; (* Quadratic Adjustment for Y *)

  kp = k1 + 1; (* direction k *)
  tmp2 = tbl[[i0 ;; i1, j0 ;; j1, km ;; kp]];
  fzz = (tmp2[[ ;; , ;; , 1]] - tmp2[[ ;; , ;; , 2]] -
      tmp2[[ ;; , ;; , 3]] + tmp2[[ ;; , ;; , 4]]) / 8.;
  fzz = fzz (1 - t^2) / 2.;
  tmp2 = Table[
    fzz[[i, j]] × wgti[[i]] × wgtj[[j]]
    , {i, 1, 2}, {j, 1, 2}];
  vzz = tmp2 // Flatten // Total; (* Quadratic Adjustment for Z *)

  value - vxx - vyy - vzz
 ];
```

In[ ]:=
```
xDom = Range[1, 141];
yDom = Range[1, 141];
zDom = Range[1, 241];

uMax = Length[xDom];
vMax = Length[yDom];
wMax = Length[zDom];

iDom = Range[uMax];
jDom = Range[vMax];
kDom = Range[wMax];
```

In[ ]:=
```
Clear[uTest, vTest, wTest];

uTest[u_] := (2 ≤ u) && (u < uMax);
vTest[v_] := (2 ≤ v) && (v < vMax);
wTest[w_] := (2 ≤ w) && (w < wMax);
```

Variables to Edit

```
fileStartAngle = 0; (*Will be appended with π/30 during runtime,
should always start at 0*)
fileEndAngle = 10; (*Will be appended with π/30 during runtime*)
Clear[allFileNames]
allFileNames = Array["" &, fileEndAngle - fileStartAngle + 1, {fileStartAngle}];
Clear[allFolderNames]
allFolderNames = Array["" &, fileEndAngle - fileStartAngle + 1, {fileStartAngle}];
Do[allFileNames[[i]] = "fitted parameters " <> ToString[i] <> "Pi to 10 Angstorms",
 {i, 1, fileEndAngle - fileStartAngle + 1}]
Do[allFolderNames[[i]] = "offCenter" <> ToString[i] <> "PiOver30",
 {i, 1, fileEndAngle - fileStartAngle + 1}]
allFileNames;
allFolderNames;
folderLocation = "ENTER ROOT FOLDER PATH HERE"
cutOffPotential = 0.5; (*in units of kt, this is the lowest
 peak amplitude that will be kept in the fitting function later*)
```

Generate cutOffPoints.wl

```
SetDirectory[]
cutOffPoints = Array[{1, 1, 0} &, {Length[allFolderNames]}];
(*{files angle, distance out, buffer for plotting}*)
Do[
 Print["Working on folder: ", allFolderNames[[folderNum]]];
 SetDirectory[folderLocation <> allFolderNames[[folderNum]]];
 allFiles = FileNames["dist=*"];
 allFiles = {allFiles[[
     Length[allFiles] - Total[StringCount[allFiles, "txt"]] + 1 ;;]], allFiles[[
      ;; Length[allFiles] - Total[StringCount[allFiles, "txt"]]]]} // Flatten;
 Do[
  Print["Working on file: ", allFiles[[fileNum]]];
  fileToCheck =
   {importPartitionButDontPutInCoordinates[folderLocation <> allFolderNames[[
        folderNum]] <> "/" <> allFiles[[fileNum]]], (folderNum - 1) / 2};
  fileToCheck = {fileToCheck};
  fullSequence =
   Map[withMaxAnglesuccessiveCirclesCenteredOnCenterToCenterAxisSamplingFunction[
      #[[1]], (fileNum - 1) * 0.5(*gapSizeInAngstroms*),
      {Pi / 120(*thetaSampleSpacing_*), Pi / 3(*maxAngleFromPole_*), 200
        (*approximateNumberOfEquatorialPhiSamples_*)}, (folderNum * Pi / 30)
      (*angleOffAxisOfCharge_*), 14.5751(*sphereRadius_*), 1.5
      (*distanceInsideOfEmbeddedCharge_*)] &, fileToCheck];
  If[Max[fullSequence // Flatten // #[[3 ;; ;; 3]] &] < cutOffPotential,
   Print["Folder ", folderNum, " has found its minimum at ", (fileNum - 1) * 0.5];
   cutOffPoints[[folderNum]] = {folderNum * Pi / 30, (fileNum + 1) * 0.5, 0};
   Break[];,
   Print["Maximum is: ", Max[fullSequence // Flatten // #[[3 ;; ;; 3]] &]];
  ]

  , {fileNum, 1, Length[FileNames[]] - 4}]

 , {folderNum, 1, 19}]
```

Generate Fitted Parameter List

This program will take the raw data compiled by the ‘Create Condensed Fitting Data’ and generate a set of parameters which characterize each fit individually. These parameters will then be used to create a general fit in the next program.

Variables to Edit

Change these to match the directory you are using as well as to match the angle range of interest.

```
angleToStartAt = 1; (*This will be prepended to Pi/30*)
angleToEndWith = 10; (*This will be prepended to Pi/30*)
distanceToReadTo = 10;
(*In Angstroms.  Must be whole numers of halves, eg. 1, 1.5, 2, 2.5...*)
folderLocation =
  "C:\\Users\\Josh\\Desktop\\Masters Research\\Two Charged Sphere Outputs
    New\\Dielectric Value of 5\\Condensed
    Interior Sphere Points Dielectric 5\\";
(*Location of condensed interior points*)


(*Protein Approximation Variables*)
protein1Radius = 14.5751;
protein1Pos = {(-12.5 - protein1Radius + 1.5) , 0, 0};
protein2Radius = 14.5751;
protein2Pos = {(-12.5 + 3) + protein2Radius - 1.5, 0, 0};

(*Generally this does not need to be edited*)
saveOutFileType = ".wl";
saveOutFileNameFormat = "Fitted Parameters ";
saveOutFolder =
  "C:\\Users\\Josh\\Desktop\\Masters Research\\Fitted Parameters\\Dielectric 5\\";
```

Functions To Load At Start

The normalize Student t-distribution. Used in all our fitting.

In[ ]:= $\texttt{normalizedStudentT[x\_, y\_, c\_ , nu\_, dof\_] :=} \left(\frac{\texttt{dof}}{\texttt{dof} + \frac{\texttt{x}^2+(-\texttt{c}+\texttt{y})^2}{\texttt{nu}^2}}\right)^{\frac{1+\texttt{dof}}{2}}$

Generate and Save Out Files

The below will generate the fitted parameter files needed to create a universal fit.

In[ ]:=
```
Do[
  saveThisOut = List[];
  Clear[condensedData];
  condensedData = Import[folderLocation <> "offCenter" <> ToString[i] <>
     "PiOver30\Condensed" <> ToString[i] <> "PiOver30All.tsv"];
  condensedData = Partition[condensedData, 3928];
```

```
Do[
 interiorY = Sin[i * Pi / 30] * (protein1Radius - 1.5);
 interiorX = Cos[i * Pi / 30] * (protein1Radius - 1.5);
 protein2NewPos = protein2Pos + {(j - 1) / 2, 0, 0};
 distance = (protein1Pos - protein2NewPos);
 distance = Sqrt[distance[[1]]^2 + distance[[2]]^2 + distance[[3]]^2];
 chargeXDistance = distance - interiorX;
 chargeHypo = Sqrt[chargeXDistance^2 + interiorY^2];
 newAngle = ArcTan[interiorY / chargeXDistance];

 maxAmp = Max[condensedData[[j]] // Transpose // #[[5]] &];

 fitTDistributionWithExponentialWeights = Map[NonlinearModelFit[#, amp * PDF[
       StudentTDistribution[0, Abs[nu], Abs[dof]], Sqrt[x^2 + (y - c)^2]],
     {{amp, 100}, {c, -(protein1Radius - 1.5) Sin[i * Pi / 30] / Pi},
      {nu, 3.7}, {dof, 2.6}}, {x, y},
     Weights → (# // Transpose // #[[3]] & // Map[Exp[#] &, #] &)] &,
   {condensedData[[j]] // Transpose // #[[3 ;; 5]] & // Transpose}];
 parameterTableData = fitTDistributionWithExponentialWeights[[1]][
     "ParameterTableEntries"] // Transpose // #[[1]] &;

 fitTDistributionWithExponentialWeights =
  Map[NonlinearModelFit[#, amp * normalizedStudentT[x, y, c, nu, dof],
     {{amp, maxAmp}, {c, -(protein1Radius - 1.5) Sin[newAngle]},
      {nu, parameterTableData[[3]]}, {dof, parameterTableData[[4]]}},
     {x, y}, Weights → (# // Transpose // #[[3]] & // Map[Exp[#] &, #] &)] &,
   {condensedData[[j]] // Transpose // #[[3 ;; 5]] & // Transpose}];

 parameterTableData = fitTDistributionWithExponentialWeights[[1]][
     "ParameterTableEntries"] // Transpose // #[[1]] &;

 parameterTableGapsFirstThenAngle =
  {(j - 1) / 2., i * Pi / 30, parameterTableData} // Flatten;

 AppendTo[saveThisOut, parameterTableGapsFirstThenAngle];

 , {j, 1, distanceToReadTo * 2 + 1}];

saveThisOut = saveThisOut // Flatten;
saveThisOut = Partition[saveThisOut, 6];
Export[saveOutFolder <> saveOutFileNameFormat <> ToString[i] <> "Pi Over 30 To " <>
  ToString[distanceToReadTo] <> " Angstroms" <> saveOutFileType, saveThisOut];
```

Generate Electrostatic Fitting Function

This generates the fit for each parameter based on the individual fit parameters generated above.

Variables to Edit

```
fileStartAngle = 0; (*Pi/30 is appended*)
fileEndAngle = 10; (*Pi/30 is appended*)
allDataFolderLocation =
  "E:\\Research Data\\Dielectric Value of 4\\Condensed Points\\";
(*Directory for the condensed data files*)
fittedParameterFileLocation = "C:\\Users\\Josh\\Desktop\\Masters
    Research\\Fitted Parameters\\Dielectric 3\\";
(*Where the previous program outputted the files*)
```

Run Before Building Functions

```
Import["E:\\Research Data\\Dielectric Value of 3\\cutoffPoints.wl"];
Clear[allFileNames]
allFileNames = Array[""&, fileEndAngle - fileStartAngle + 1, {fileStartAngle}];
Do[allFileNames[[i + 1]] =
  "Fitted Parameters " <> ToString[i] <> "Pi Over 30 To 10 Angstroms",
 {i, fileStartAngle, fileEndAngle - fileStartAngle}]
```

Build Weighting Function

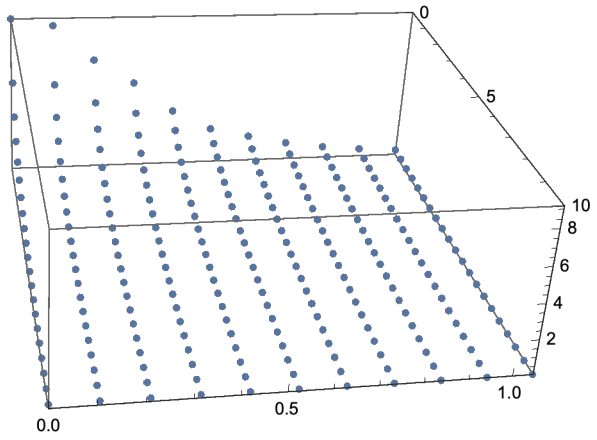
This constructs an exponential weight for the data based on the potential observed at the peak for each relative orientation.

```
weightsBasedOnExpPotential = List[];
Do[
  If[cutOffPoints[[i]][[2]] > 0
   ,
   condensedData = Import[allDataFolderLocation <> "offCenter" <> ToString[i - 1] <>
       "PiOver30\Condensed" <> ToString[i - 1] <> "PiOver30All.tsv"];
  ];
  Do[
   maxVal = Max[condensedData[[3928 * (j - 1) + 1 ;; (j) * 3928, 5]]];
   maxVal = Exp[maxVal];
   AppendTo[weightsBasedOnExpPotential, maxVal]
   , {j, 1, cutOffPoints[[i]][[2]] * 2}]
  , {i, 1, Length[cutOffPoints]}];
weightsBasedOnExpPotential;
```

```
In[•]:= saveData = Array[0 &, {Length[allFileNames], 3}];
Do[
 fullData =
  Import[fittedParameterFileLocation <> ToString[allFileNames[[i + 1]]] <> ".wl"];
 fullData = fullData // Flatten // Partition[#, 6] & // Transpose;
 fullData[[2]] = Array[i * Pi / 30 &, Length[fullData[[1]]]];
 fullData = fullData // Transpose;
 saveData[[i + 1]] = fullData;
 , {i, fileStartAngle, fileEndAngle}]
```

```
In[•]:= plotThis = saveData // Flatten // Partition[#, 6] & // Transpose //
     {#[[1]], #[[2]], #[[3]]} & // Transpose;
ListPointPlot3D[plotThis, PlotRange → All]
```

Out[•]=



Build data to fit for each parameter

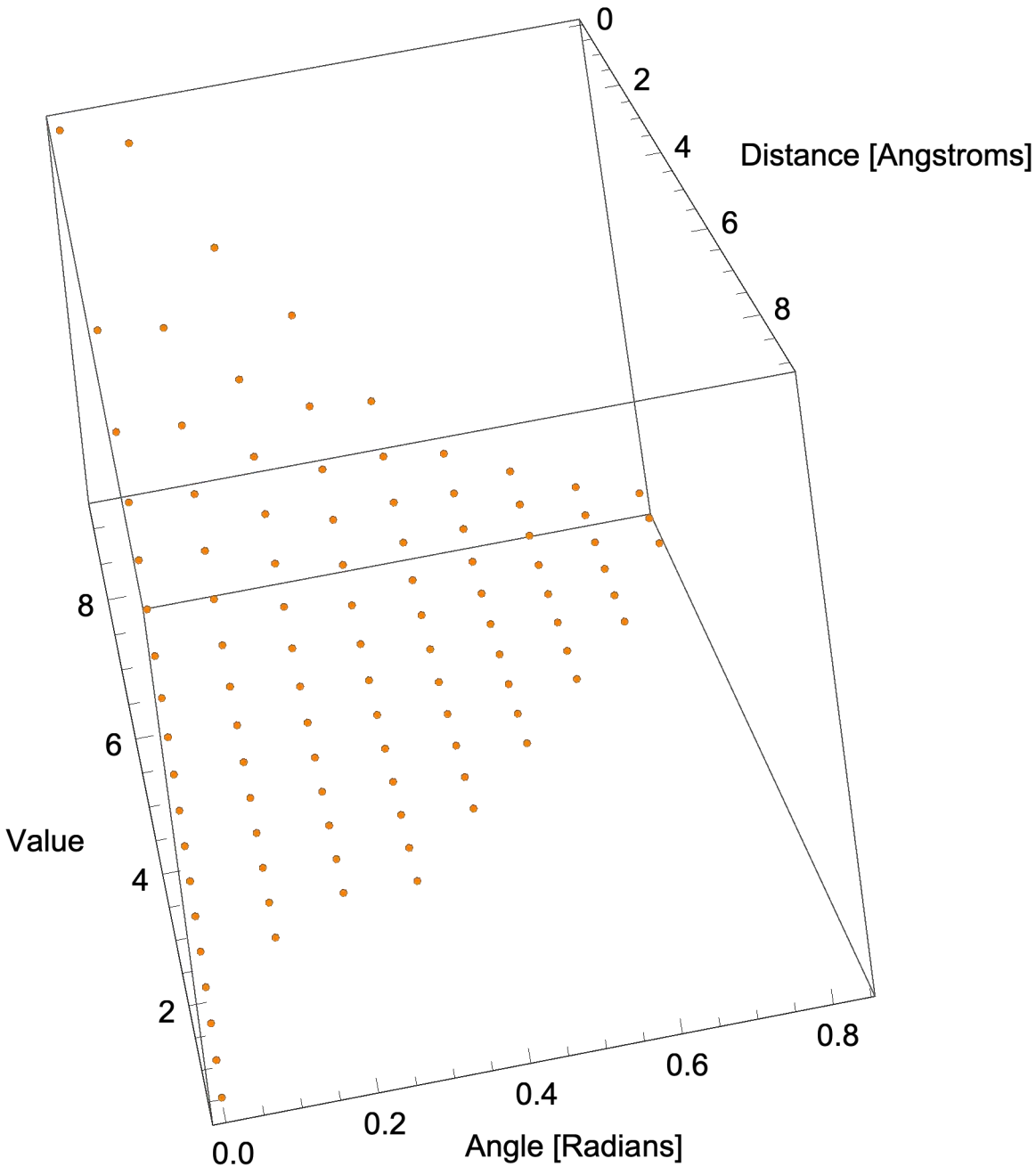
This creates a series of arrays which contain all the parameters needed to characterize the individual relative orientations. This will be processed below to create a universal fit.

```
newDataStuff = List[];
rowLength = Length[saveData];
colLength = Length[saveData[[1]]];
Do[
 tempData = saveData[[i]];
 If[cutOffPoints[[i]][[2]] ≠ 0,
  AppendTo[newDataStuff, tempData[[1 ;; cutOffPoints[[i]][[2]] * 2 ]]];
 ]
 , {i, 1, Length[saveData]}]
newDataStuff = newDataStuff // Flatten // Partition[#, 6] & // Transpose;
ampData = {newDataStuff[[1]], newDataStuff[[2]], newDataStuff[[3]]} // Transpose;
centerData =
  {newDataStuff[[1]], newDataStuff[[2]], newDataStuff[[4]]} // Transpose;
nuData = {newDataStuff[[1]], newDataStuff[[2]], newDataStuff[[5]]} // Transpose;
dofData = {newDataStuff[[1]], newDataStuff[[2]], newDataStuff[[6]]} // Transpose;
```

Build Amplitude Fit

In[•]:=
```
amplitudeFitPlot =
  ampData // Map[({PointSize[.01], Orange, Point[#]}) &, #] & // Graphics3D //
   Show[#, BoxRatios -> {2, 1, 1}, Axes → True,
     AxesLabel → {"Distance [Angstroms]", "Angle [Radians]", "Value"},
     BaseStyle → {FontSize -> 18}, ImageSize → 600] &;
Show[amplitudeFitPlot]
```

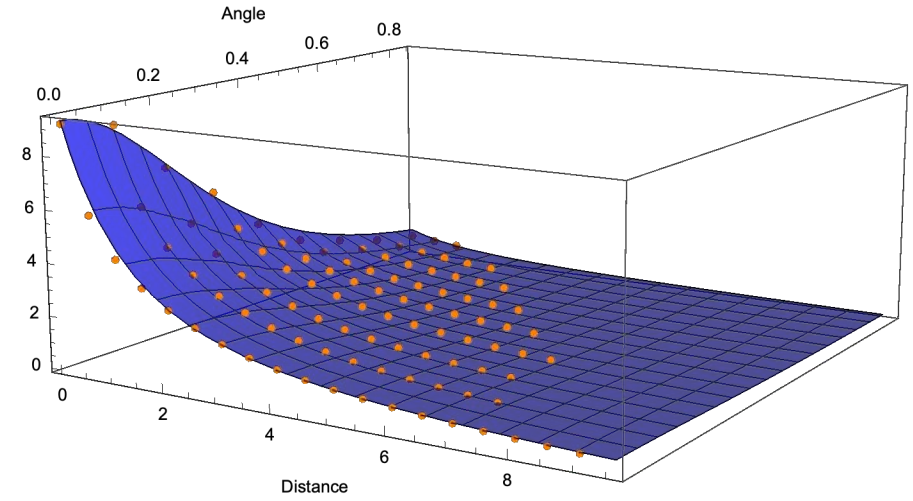
Out[•]=



In[•]:=
```
normalizedOneDStudentT[ang_, k2_, k3_] :=
  (1 / PDF[StudentTDistribution[0, k2, k3], 0]) *
   PDF[StudentTDistribution[0, k2, k3], ang]
```

In[•]:=
```
ampFoundFit = NonlinearModelFit[ampData,
    amp * normalizedOneDStudentT[ang, k2, k3] * (1 / (3. + dist) ^ k1),
    {{amp, 6000}, {k1, 1.95}, {k2, 0.355}, {k3, 1.9}},
    {dist, ang}, Weights → Sqrt[weightsBasedOnExpPotential]];
Show[Plot3D[ampFoundFit // Normal, {dist, 0, 9.55}, {ang, 0, 8 Pi / 30},
  PlotRange → All, AxesLabel → {"Distance", "Angle", "eV/kT"}, ImageSize → 500,
  PlotStyle → Directive[Blue, Opacity[0.7]]], amplitudeFitPlot]
ampFoundFit // Normal
```

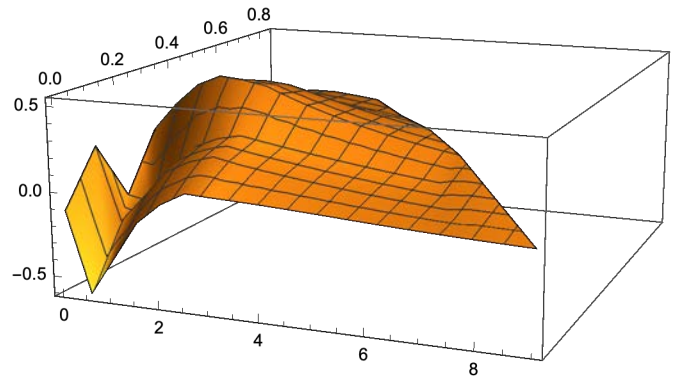
Out[•]=



Out[•]=
$$\frac{248.917 \left(\frac{1}{1.7117+9.96102\ \text{ang}^2}\right)^{1.35585}}{(3.+\text{dist})^{2.33009}}$$

In[•]:=
```
residuals = ampData // Transpose //
    {#[[1]], #[[2]], ampFoundFit["FitResiduals"]} & // Transpose;
ListPlot3D[residuals]
```

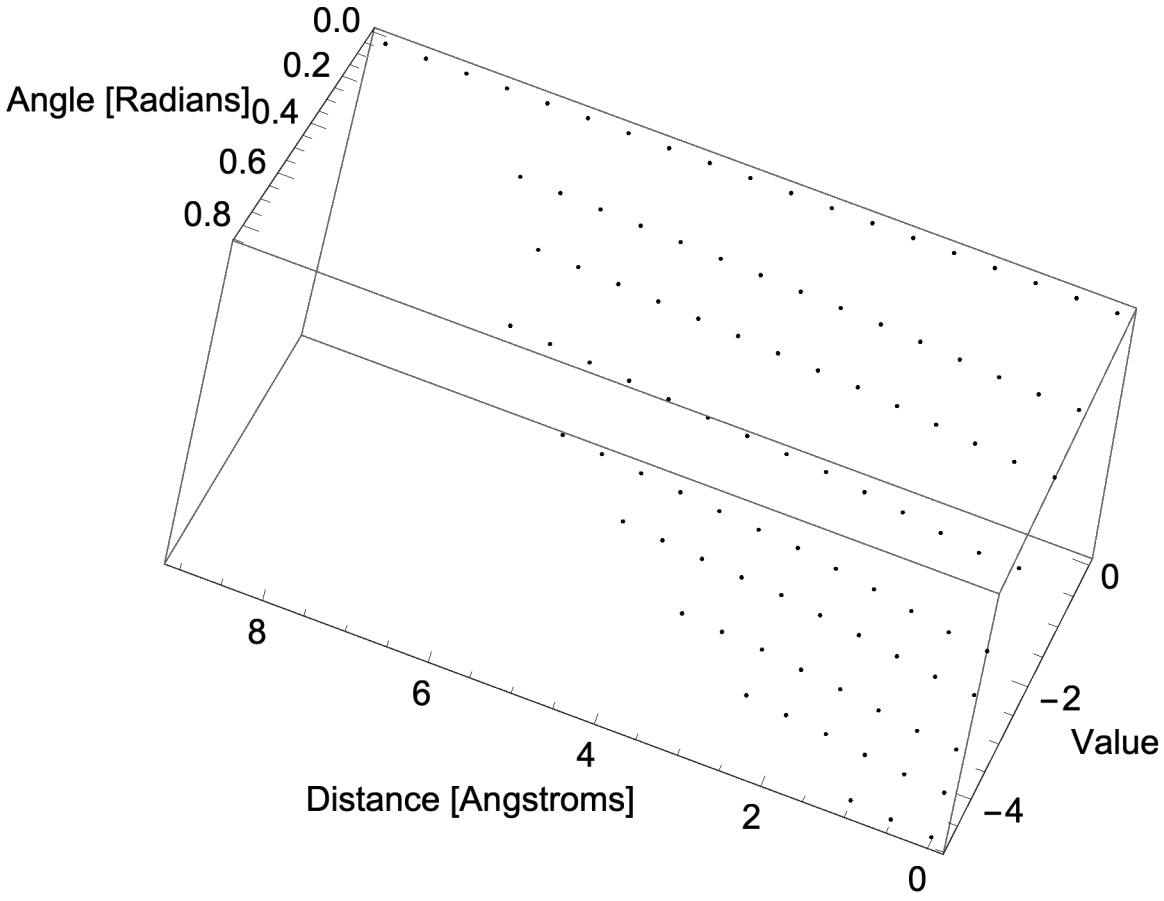
Out[•]=



Plot and Fit for Center

In[•]:=
```
centerFitPlot =
 centerData // Map[({PointSize[.005], Black, Point[#]}) &, #] & // Graphics3D //
  Show[#, BoxRatios -> {2, 1, 1}, Axes → True,
    AxesLabel → {"Distance [Angstroms]", "Angle [Radians]", "Value"},
    BaseStyle → {FontSize -> 18}, ImageSize → 600] &
```

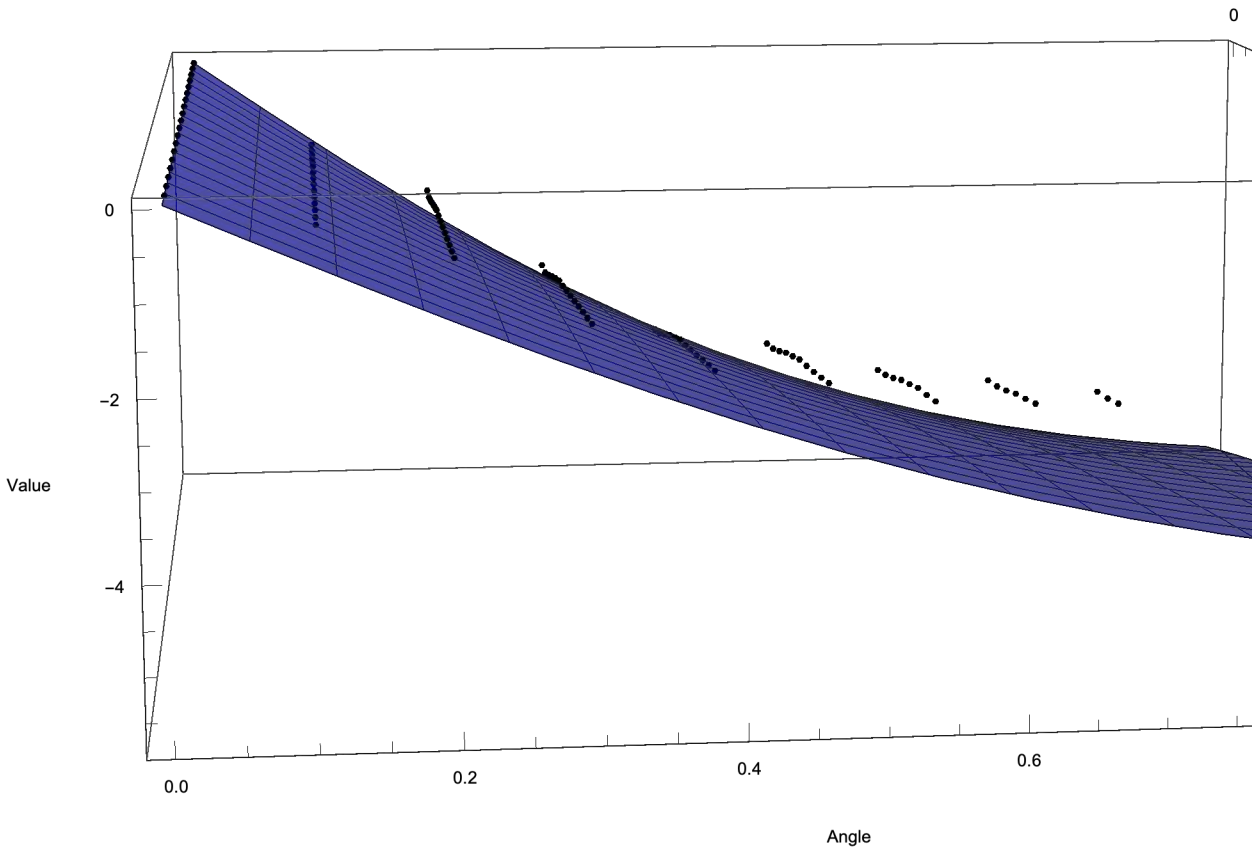
Out[•]=



```
In[•]:= centerFitNew[gap_, angle_] :=
 Module[{newAngle = angle, protein1Radius, protein1Pos, protein2Radius, protein2Pos,
   protein2NewPos, interiorY, interiorX, distance, chargeXDistance, chargeHypo},
  protein1Radius = 14.5751;
  protein1Pos = {(-12.5 - protein1Radius + 1.5) , 0, 0};
  protein2Radius = 14.5751;
  protein2Pos = {(-12.5 + 3) + protein2Radius - 1.5, 0, 0};
  protein2NewPos = protein2Pos + {gap, 0, 0};
  interiorY = Sin[angle] * (protein1Radius - 1.5);
  interiorX = Cos[angle] * (protein1Radius - 1.5);
  distance = (protein1Pos - protein2NewPos);
  distance = Sqrt[distance[[1]]^2 + distance[[2]]^2 + distance[[3]]^2];
  chargeXDistance = distance - interiorX;
  chargeHypo = Sqrt[chargeXDistance^2 + interiorY^2];
  newAngle = ArcTan[interiorY / chargeXDistance];
  Return[-(protein2Radius - 1.5) * Sin[newAngle]]]
```

```
In[•]:= Show[Plot3D[centerFitNew[dist, ang], {dist, 0, 9.5}, {ang, 0 Pi / 30, 9 Pi / 30},
   PlotRange → All, AxesLabel → {"Distance", "Angle", "Value"},
   ImageSize → 800, PlotStyle → Directive[Blue, Opacity[0.7]]], centerFitPlot]
```

Out[•]=



Plot and Fit for Nu

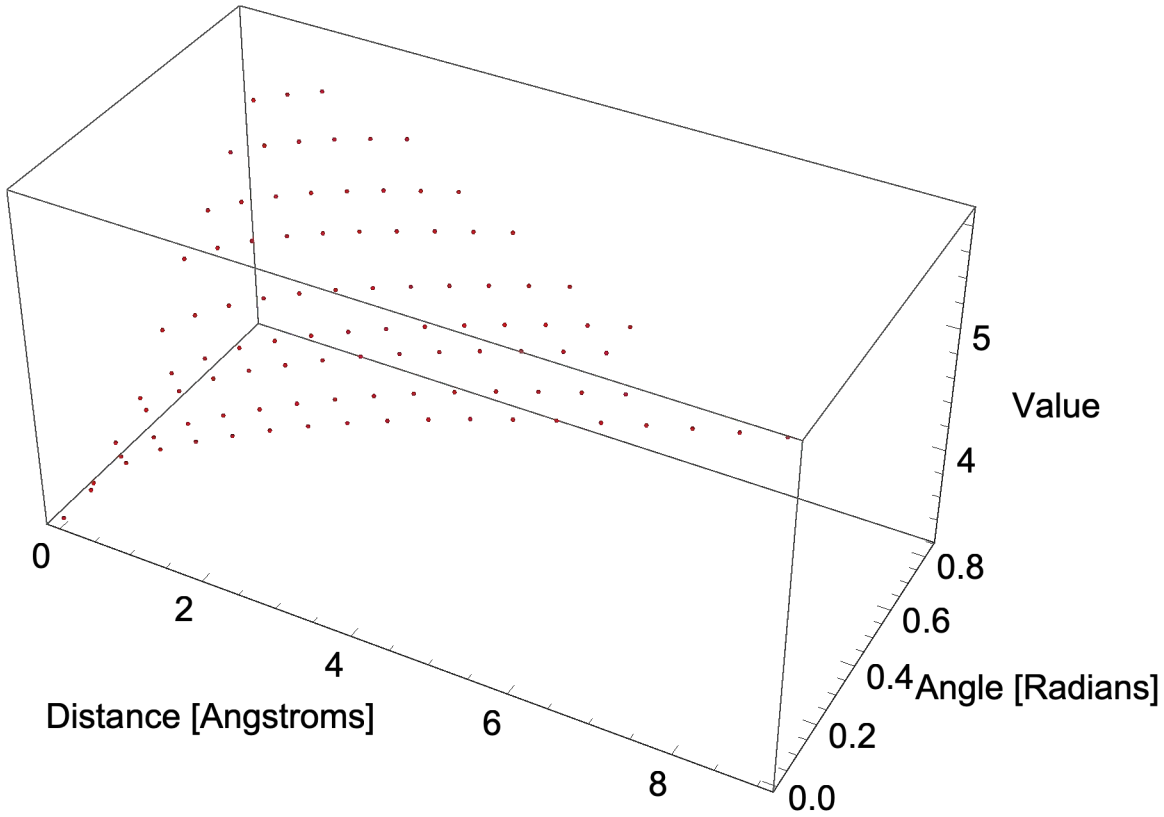
```
In[•]:= nuFoundFit = NonlinearModelFit[nuData // Flatten // Partition[#, 3] &,
          a1 * ang^2 + a2 * ang + nuData[[1, 3]] + b1 * dist^2 + b2 * dist + ab * dist * ang,
          {{a1, 1}, {a2, 2}, {b1, .25}, {b2, -.35}, {ab, 1.3}},
          {dist, ang}, Weights → weightsBasedOnExpPotential]
        nuFoundFit // Normal
```

```
Out[•]= FittedModel[ 3.25556 + 0.0042507 ang + 4.15246 ang^2 + ≪19≫ ≪4≫ - 0.547384 ang dist - 0.0521855 dist^2 ]
```

```
Out[•]= 3.25556 + 0.0042507 ang + 4.15246 ang^2 + 0.720485 dist - 0.547384 ang dist - 0.0521855 dist^2
```

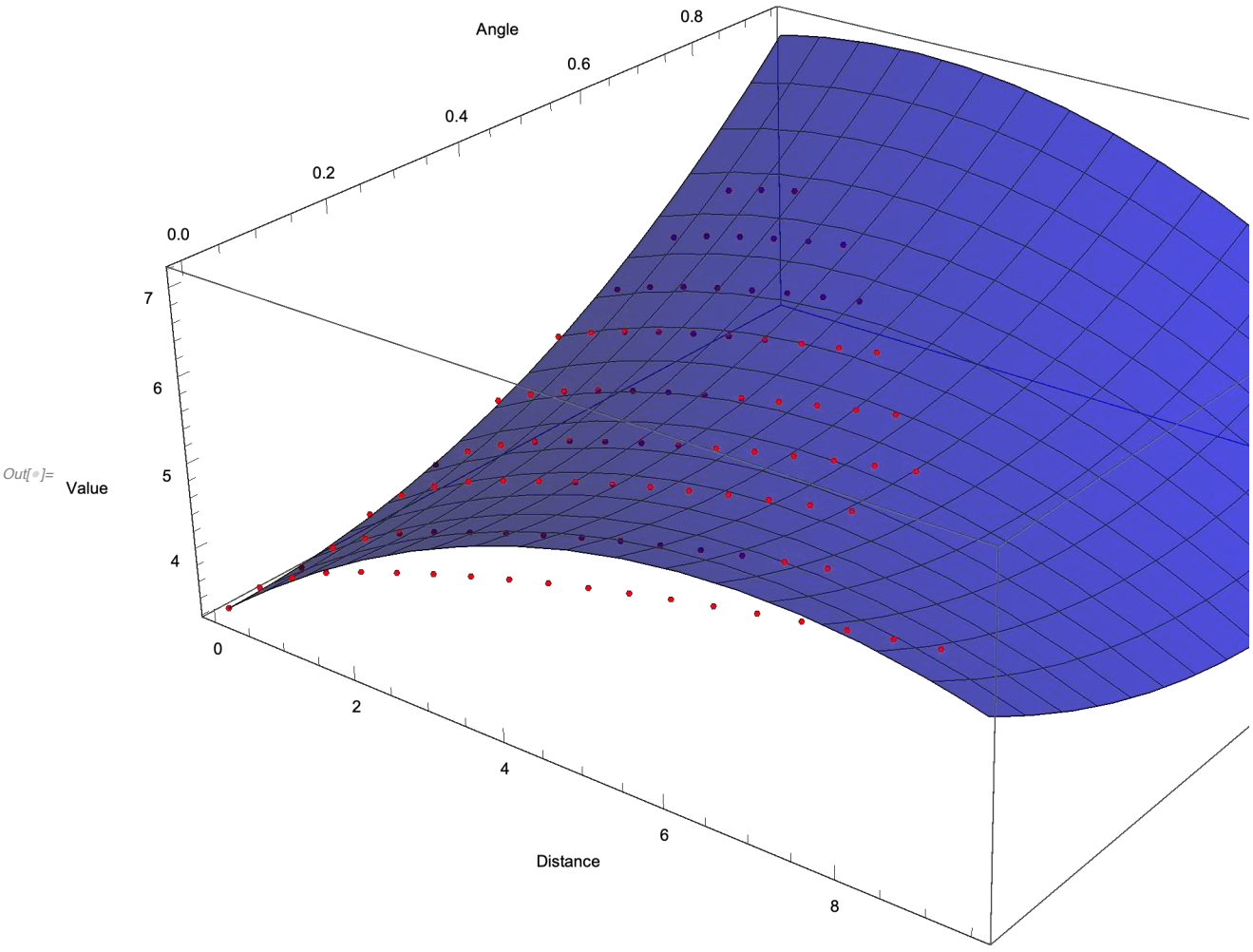
```
In[•]:= nuFitPlot = nuData // Map[({PointSize[.005], Red, Point[#]}) &, #] & // Graphics3D //
          Show[#, BoxRatios -> {2, 1, 1}, Axes → True,
            AxesLabel → {"Distance [Angstroms]", "Angle [Radians]", "Value"},
            BaseStyle → {FontSize -> 18}, ImageSize → 600] &
```

```
Out[•]=
```



```
In[•]:= Show[Plot3D[nuFoundFit // Normal, {dist, 0, 9.5}, {ang, 0 Pi / 30, 9 Pi / 30},
  PlotRange → All, AxesLabel → {"Distance", "Angle", "Value"},
  ImageSize → 800, PlotStyle → Directive[Blue, Opacity[0.7]]], nuFitPlot]
```

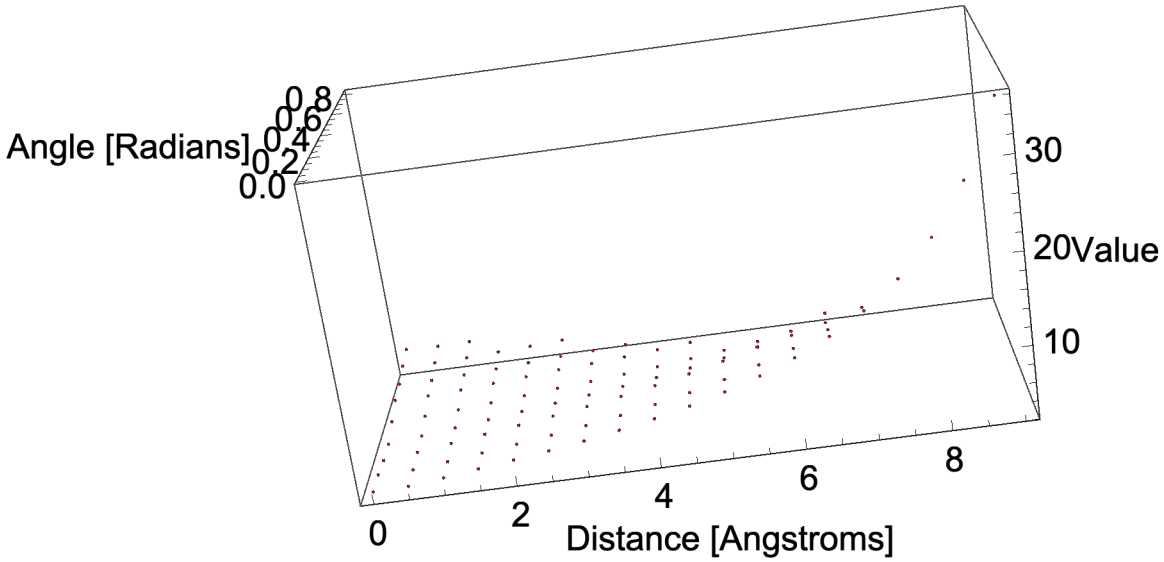
Out[•]=



Plot and Fit for Degrees of Freedom

In[•]:=
```
dofFitPlot = dofData // Map[({PointSize[.005], Red, Point[#]}) &, #] & // Graphics3D //
  Show[#, BoxRatios -> {2, 1, 1}, Axes → True,
    AxesLabel → {"Distance [Angstroms]", "Angle [Radians]", "Value"},
    BaseStyle → {FontSize -> 18}, ImageSize → 600] &
```

Out[•]=



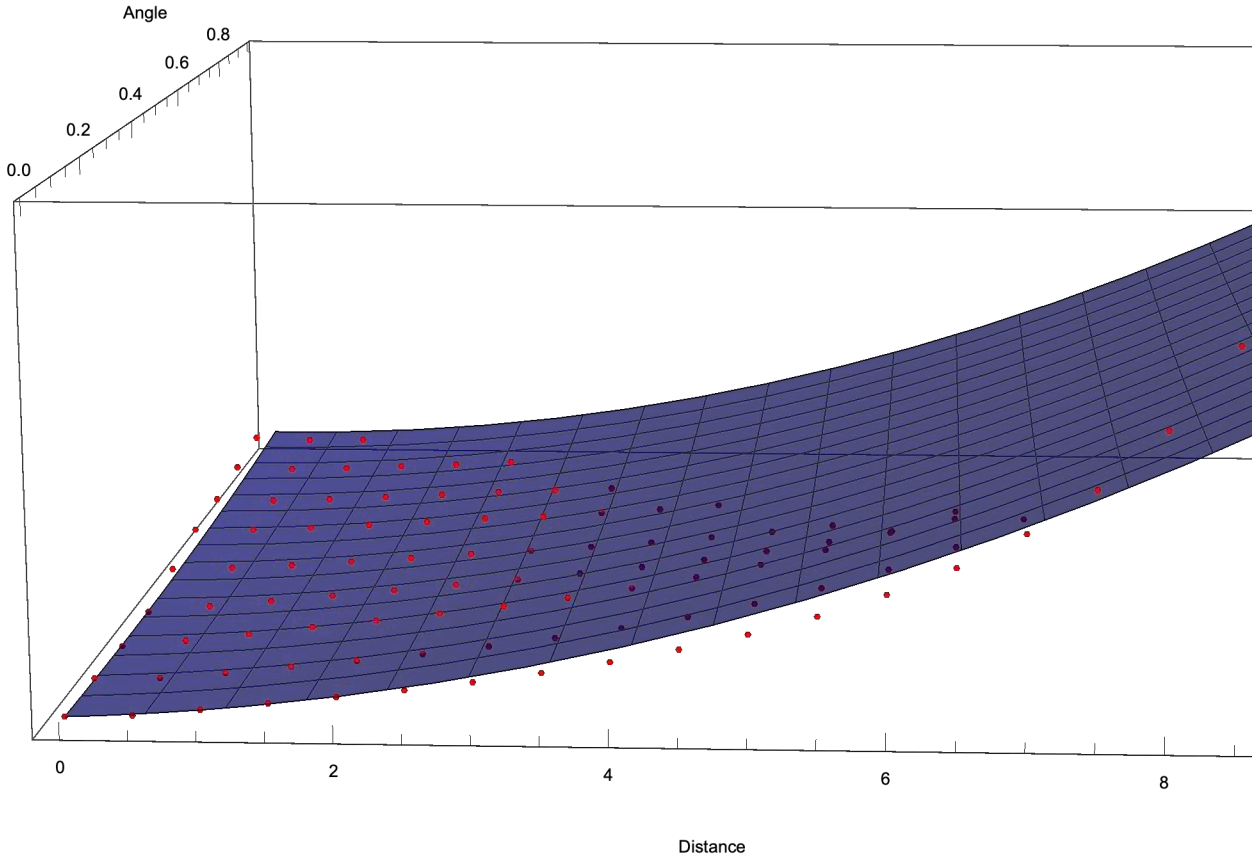
In[•]:=
```
dofFoundFit = NonlinearModelFit[dofData // Flatten // Partition[#, 3] &,
  a1 * ang^2 + a2 * ang + dofData[[1]][[3]] + b1 * dist^2 + b2 * dist + ab * ang * dist,
  {{a1, 1}, {a2, 2}, {b1, .25}, {b2, -.35}, {ab, 0}},
  {dist, ang}, Weights → weightsBasedOnExpPotential]
dofFoundFit // Normal
```

Out[•]= FittedModel[2.96466 − 1.66685 ang + 2.66387 ang^2 + ≪20≫ dist − 0.449697 ang dist + 0.219881 dist^2]

Out[•]= $2.96466 - 1.66685\ \text{ang} + 2.66387\ \text{ang}^2 + 0.339181\ \text{dist} - 0.449697\ \text{ang}\ \text{dist} + 0.219881\ \text{dist}^2$

In[•]:=
```
Show[Plot3D[dofFoundFit // Normal, {dist, 0, 9.5}, {ang, 0 Pi / 30, 9 Pi / 30},
  PlotRange → All, AxesLabel → {"Distance", "Angle", "Value"},
  ImageSize → 800, PlotStyle → Directive[Blue, Opacity[0.7]]], dofFitPlot]
```

Out[•]=



Maximum Error of All Orientations and distances

Find the maximum difference between the fit and the data exported from the PDE solver.

Variables to Edit

```
(*Stuff to Edit*)
rootDirectory = "E:\\Research Data\\Dielectric Value of 3\\Condensed Points\\";
(*Root directory of the condensed data points*)
```

Functions To Load At Start

In[•]:=
```
studentTRescale[v_, dof_] := 1 / PDF[StudentTDistribution[0, v, dof], 0]
```

```
normalizedStudentT[x_, y_, c_ , nu_, dof_] := (dof / (dof + (x^2 + (-c + y)^2) / nu^2))^((1 + dof) / 2)
```

```
In[•]:= fullFunctionAssembled[x_, y_, angle_, gap_] :=
          ampFoundFit[gap, angle] * normalizedStudentT[x, y,
            centerFitNew[gap, angle] , nuFoundFit[gap, angle], dofFoundFit[gap, angle]];
```

Generate

```
In[•]:= tempData = {};
        trueFailure = {0, 0, 0, 0, 0, 0};
        Do[
         condensedData = Import[rootDirectory <> "\\offCenter" <> ToString[angle] <>
            "PiOver30\Condensed" <> ToString[angle] <> "PiOver30All.tsv"];
         (*mapCondensedData = Partition[condensedData[[
             1;;((cutOffPoints[[angle+1]][[2]] * 2)+1)*3928,3;;5]],3929];*)
         mapCondensedData = Partition[condensedData[[ ;; , 3 ;; 5]], 3929];

         Do[

          workingData = mapCondensedData[[distance + 1]];
          transposedData = workingData // Transpose;

          workingBiggestFailure = {0, 0, 0, 0};

          Do[
           calcDataPoint = fullFunctionAssembled[transposedData[[1]][[i]],
             transposedData[[2]][[i]], angle * Pi / 30, distance];
           trueDataPoint = transposedData[[3]][[i]];
           difference = calcDataPoint - trueDataPoint;
           percentDiff = calcDataPoint / trueDataPoint;

           If[ Abs[difference] > Abs[ workingBiggestFailure[[1]] ],
            workingBiggestFailure =
             {difference, percentDiff, calcDataPoint, trueDataPoint};
            If[Abs[difference] > Abs[trueFailure[[3]]],
             trueFailure = {angle, distance / 2,
               difference, percentDiff, calcDataPoint, trueDataPoint};
             Print["New large difference", trueFailure]
             ,
             Nothing]
           ,
           Nothing]

          , {i, 1, 3929}];
```

```
    AppendTo[tempData, {angle, (distance) / 2, workingBiggestFailure} // Flatten];

    , {distance, 0, cutOffPoints[[angle + 1]][[2]] * 2}]
  , {angle, 0, 8}]
Print[trueFailure];
Print[tempData];
```

Visual Representation of Single Relative Orientation and Fit

Generate a visual model of the fit for any relative orientation. Useful for getting an idea as to the quality of the general fit.

Variables to Edit

```
(*Stuff to Edit*)
distanceToLookAt = 0; (*Between 0 and 10,
whole numbers or halves only, eg. 3/2*)
angleToLookAt = 1; (*whole numbers only*)
condensedData = "E:\\Research Data\\Dielectric Value of 3\\Condensed Points";
Import[condensedData <> "\\offCenter" <> ToString[angleToLookAt] <>
   "PiOver30\Condensed" <> ToString[angleToLookAt] <> "PiOver30All.tsv"];
(*This is where the condensed points are stored*)

(*Protein data.  Generally it doesn't need to be changed*)
protein1Radius = 14.5751;
protein1Pos = {(-12.5 - protein1Radius + 1.5) , 0, 0};
protein2Radius = 14.5751;
protein2Pos = {(-12.5 + 3) + protein2Radius - 1.5, 0, 0};
```

Functions To Load At Start

```
studentTRescale[v_, dof_] := 1 / PDF[StudentTDistribution[0, v, dof], 0]

normalizedStudentT[x_, y_, c_ , nu_, dof_] := (dof / (dof + (x^2 + (-c+y)^2) / nu^2))^((1+dof)/2)

fullFunctionAssembled[x_, y_, angle_, gap_] :=
  ampFoundFit[gap, angle] * normalizedStudentT[x, y,
    centerFitNew[gap, angle] , nuFoundFit[gap, angle], dofFoundFit[gap, angle]];
```

Generate Visual Representation

```
(*Fixed Procedures, do not touch*)

proteinOpacity = 0.3;
displayThisData = List[];
graphExtent = 12.0;
protein2NewPos = protein2Pos + {distanceToLookAt, 0, 0};
Do[
 AppendTo[displayThisData,
  {protein2NewPos[[1]] - Sqrt[(protein2Radius - 1.5)^2 - condensedData[[i, 3]]^2 -
      condensedData[[i, 4]]^2], protein2NewPos[[2]] -
    condensedData[[i, 4]], protein2NewPos[[3]] - condensedData[[i, 3]]}]
 , {i, 1, 3929}]
displayThisData;


(*Calculations for drawing informational lines*)
interiorY = Sin[angleToLookAt * Pi / 30] * (protein1Radius - 1.5)
interiorX = Cos[angleToLookAt * Pi / 30] * (protein1Radius - 1.5)
distance = (protein1Pos - protein2NewPos);
distance = Sqrt[distance[[1]]^2 + distance[[2]]^2 + distance[[3]]^2];
chargeXDistance = distance - interiorX;
chargeHypo = Sqrt[chargeXDistance^2 + interiorY^2];
newAngle = ArcTan[interiorY / chargeXDistance]
chargeLocation = protein1Pos + {(protein1Radius - 1.5) Cos[angleToLookAt * Pi / 30],
    (protein1Radius - 1.5) Sin[angleToLookAt * Pi / 30], 0};

(*Creating graphs*)
theThing = Show[
   Graphics3D[{Cyan, Opacity[proteinOpacity], Sphere[protein1Pos, protein1Radius]}],
   Graphics3D[{Blue, Opacity[proteinOpacity],
     Sphere[protein2NewPos, protein2Radius]}], Graphics3D[{Cyan,
     Opacity[proteinOpacity + 0.2], Sphere[protein1Pos, protein1Radius - 1.5]}],
   Graphics3D[{Blue, Opacity[proteinOpacity + 0.2],
     Sphere[protein2NewPos, protein2Radius - 1.5]}],
   ListPointPlot3D[{chargeLocation}, PlotStyle → Red],
   ListPointPlot3D[displayThisData, PlotStyle → {Orange, PointSize[0.005]}],
   Graphics3D[Line[{protein1Pos, protein2NewPos}]],
   Graphics3D[Line[{protein1Pos, chargeLocation}]],
   Graphics3D[Line[{protein1Pos + {interiorX, 0, 0}, chargeLocation}]],
   Graphics3D[Line[{protein2NewPos, chargeLocation}]],
    ImageSize → 900];

workingModel = Show[ListPointPlot3D[condensedData[[
```

```
      3928 * distanceToLookAt / 0.5 + 1 ;; (((distanceToLookAt) / 0.5 ) + 1) * 3928,
      3 ;; 5]], PlotStyle → Orange, PlotRange → All],
    Plot3D[fullFunctionAssembled[x, y, angleToLookAt * Pi / 30, distanceToLookAt],
     {x, -graphExtent, graphExtent}, {y, -graphExtent, graphExtent},
     PlotRange → All, PlotStyle → {Blue, Opacity[proteinOpacity + 0.2]}],
    Graphics3D[{Thick, Red, Line[{{0, -(protein2Radius - 1.5) * Sin[newAngle], 0},
        {0, -(protein2Radius - 1.5) * Sin[newAngle],
         Max[condensedData[[3928 * distanceToLookAt / 0.5 + 1 ;;
             (((distanceToLookAt) / 0.5 ) + 1) * 3928, 5]]] * 1.2}}]}]];
GraphicsColumn[{theThing, workingModel}, ImageSize → 900]
```

The Quadratic Interpolator

The quadratic interpolator used was written by Dr. J. Hamilton. The first section deals with defining the region the interpolator is going to work over. It is important that 'xDom', 'yDom' and 'zDom' are set to have the same range as the 'tbl' passed to 'quadraticInterpolator'. From this the maximum iterator value is deduced as the length of each dimension of the region. Then by inputting this into the Range[] function we are able to recast a range (which might not start at 1) into a domain which can be iterated over safely.

```
xDom = Range[1, 141];
yDom = Range[1, 141];
zDom = Range[1, 241];

uMax = Length[xDom];
vMax = Length[yDom];
wMax = Length[zDom];

iDom = Range[uMax];
jDom = Range[vMax];
kDom = Range[wMax];

Clear[uTest, vTest, wTest];

uTest[u_] := (2 ≤ u) && (u < uMax);
vTest[v_] := (2 ≤ v) && (v < vMax);
wTest[w_] := (2 ≤ w) && (w < wMax);
```

The above 'Test' functions are designed to see if the input points for the interpolator are on the edge of the region. If they are the interpolator is unable to interpolate there because the derivative is undefined in the adjacent cell's face.

```
quadraticInterpolator[u_, v_, w_, tbl_] :=
 Module[{okay, i0, j0, k0, i1, j1, k1, r, s, t, im, jm, km, ip, jp, kp,
   wgti, wgtj, wgtk, value, tmp1, tmp2, fxx, vxx, fyy, vyy, fzz, vzz},
  okay = uTest[u] && vTest[v] && wTest[w];
  If[! okay, Return["Domain Fault"]];
```

u,v,w are reals, they are the x,y,z coordinates of the point we are trying to interpolate for.

The below gives us the integer which designates the lowermost corner of the box our interpolation point resides within.

```
i0 = u // Floor;
j0 = v // Floor;
k0 = w // Floor;
```

”Floor” Produces the largest integer which is less than or equal to the value, essentially this is a strict round down.

The below gives us the integer which designates the uppermost corner of the box our interpolation point resides within.

```
i1 = i0 + 1;
j1 = j0 + 1;
k1 = k0 + 1;
```

This takes the decimal of the real you started with (from u-i0) and rescales it to be between in the range (-1,1).

```
r = 2 (u – i0) – 1;
s = 2 (v – j0) – 1;
t = 2 (w – k0) – 1;

(*This gives us the index for the cells below (on each axis) in our array*)
im = i0 – 1;
jm = j0 – 1;
km = k0 – 1;
```

Gives us the averaged weight in one dimension as a list of length 2. The first entry is the weight of the left-hand (i0) part of the table and the second entry is the right-hand side of the table. This is a linear weight based on distance from edges of the cell in question for the interpolation. Note that the sum of the weights is always one (r’s cancel)

```
wgti = {1 – r, 1 + r} / 2.;
wgtj = {1 – s, 1 + s} / 2.;
wgtk = {1 – t, 1 + t} / 2.;

(* TriLinear Value *)
tmp1 = Table[
   tbl[[im + i, jm + j, km + k]] × wgti[[i]] × wgtj[[j]] × wgtk[[k]]
   , {i, 1, 2}, {j, 1, 2}, {k, 1, 2}];
```

The above has 2^3 entries (8) and note that because of mathematica not allowing indexing starting at 0 we have to start one below the index of the cell we’ re interested in. This is the origin of the ‘ im’, ‘ jm’, ‘ km’ in tbl[[]]. The outputs of the table are the vertices of the cube modified by the distance from which you are from the sides of the cell (coming from the weights ‘ wgti’, ‘ wgtj’, ‘ wgtk’).

```
value = tmp1 // Flatten // Total;
```

Sum all the values to get the linear interpolation

```
(* Second Order Differences and Quadratic Adjustments *)
ip = i1 + 1;(* direction i *)
tmp2 = tbl[[im ;; ip, j0 ;; j1, k0 ;; k1]];
```

This is a slice from the array we're interpolating on extended only in one dimension. The indexes are for the vertices of the cells! This gives us 3 adjacent cells in the i-direction

```
fxx = (tmp2[[1, ;; , ;;]] -
      tmp2[[2, ;; , ;;]] - tmp2[[3, ;; , ;;]] + tmp2[[4, ;; , ;;]]) / 8.;
```

Above we've started with the four vertices in the jk plane and starting from the low side subtract the second set of four vertices, the third set, and then added the fourth set.

```
fxx = fxx (1 - r^2) / 2.;
```

By multiplying by this you are forcing the second derivative to have the proper weight; which is to say with this addition you can make the interpolation quadratic (in one dimension for now) after weighting it properly.

```
tmp2 = Table[
   fxx[[j, k]] × wgtj[[j]] × wgtk[[k]]
   , {j, 1, 2}, {k, 1, 2}];
```

This is looking end on at the 3 cubes we were working with before. We now weight the parabolic interpolation linearly along the other two dimensions (not x).

```
vxx = tmp2 // Flatten // Total;(* Quadratic Adjustment for X *)
```

The below is the same as above just for the other two dimensions.

```
jp = j1 + 1; (* direction j *)
tmp2 = tbl[[i0 ;; i1, jm ;; jp, k0 ;; k1]];
fyy = (tmp2[[ ;; , 1, ;;]] -
      tmp2[[ ;; , 2, ;;]] - tmp2[[ ;; , 3, ;;]] + tmp2[[ ;; , 4, ;;]]) / 8.;
fyy = fyy (1 - s^2) / 2.;
tmp2 = Table[
   fyy[[i, k]] × wgti[[i]] × wgtk[[k]]
   , {i, 1, 2}, {k, 1, 2}];
vyy = tmp2 // Flatten // Total; (* Quadratic Adjustment for Y *)

kp = k1 + 1; (* direction k *)
tmp2 = tbl[[i0 ;; i1, j0 ;; j1, km ;; kp]];
fzz = (tmp2[[ ;; , ;; , 1]] -
      tmp2[[ ;; , ;; , 2]] - tmp2[[ ;; , ;; , 3]] + tmp2[[ ;; , ;; , 4]]) / 8.;
fzz = fzz (1 - t^2) / 2.;
tmp2 = Table[
   fzz[[i, j]] × wgti[[i]] × wgtj[[j]]
   , {i, 1, 2}, {j, 1, 2}];
vzz = tmp2 // Flatten // Total;(* Quadratic Adjustment for Z *)
```

The below is the linear interpolation minus the quadratic corrections. They are minus because the parabola defined by the quadratic adjustment is ‘pinned’ at the datapoints. Since if the parabola is concave up it must hang below the linear term we subtract it off from the linear interpolation.

```
value - vxx - vyy - vzz
  ];
```